Performance improvements and future language extensions in the pqR implementation of R

Radford M. Neal, University of Toronto

Dept. of Statistical Sciences and Dept. of Computer Science

http://www.cs.utoronto.ca/∼radford

http://radfordneal.wordpress.com

http://pqR-project.org

Greater Toronto Area R User Group, 27 April 2016

Why I Decided to Create pqR

When R first came out, I was delighted that its implementation was far better than that of S. I didn't look into the details.

But in August 2010 I happened to discover two things about R-2.11.1:

- {`a+b`}/{`a*b`} was faster than `(a+b)/(a*b)` (when `a` and `b` are scalars).
- `a*a` was faster than `a^2` (when `a` is a long vector).

I realized that there was much "low hanging fruit" in the R interpreter, and made patches to R-2.11.1 which sped up parentheses, squaring, and several other operations, including reducing general overhead.

A few of my patches were incorporated into R-2.12.0, but R Core was uninterested in most of them — eg, a small patch that speeds up matrix-vector multiplies by a factor of five (still not adopted more than five years later).

Current Status of pqR

After expanding my speed patches considerably, and implementing the major feature of automatically utilizing multiple cores, I released the first version of pqR, a "pretty quick" R, in June 2013.

pqR also fixes some bugs present in R Core versions, and has some new features, such as better tracing of memory usage.

The current version is pqR-2015-09-14, available from `pqR-project.org`.

Some improvements from pqR have been put into R-3.1.0 or later R core versions, but most have not. This includes some changes that would produce large performance improvements, with very little effort, as well as some significant bug fixes, and significant code cleanups.

Compatibility and Installation of pqR

pqR is based on R-2.15.0, with some features and bug fixes from later R Core versions.

pqR is compatible with almost all packages that work with R-2.15.0.
Some packages needing features from later R Core versions that are also in pqR should also run (if their dependency on the R version is changed).

pqR has been tested and works in the following environments:

- Linux systems, with Intel/AMD (32 or 64 bit), ARM, or PowerPC processors.
- Mac OS X systems, with Intel (32 or 64 bit) processors.
- Windows 7/8/8.1/10 systems, with Intel/AMD (32 or 64 bit) processors.
- Solaris 10 and Solaris 11 systems, with SPARC processors.

pqR is currently distributed only in source form, but pre-compiled binary versions may be released soon.

Performance Improvements in pqR

Partial List of Performance Improvements

- Detailed code improvements — local changes to the interpreter, fixing inefficient code. Some code in the R interpreter is also badly-written in other respects.
- Better reference counting, so that objects are duplicated less often. For example, `a<-1:1000; b<-a; a<-2:1000; b[1:10]<-0:9` shouldn't require duplication for the assignment to `b[1:10]`.
- Avoidance of unnecessary memory allocations for temporary results. For example, `a<-rnorm(1000); b<-(a+1)/2` should not require allocating separate storage for `a+1`. (Incorporated into R-3.1.0.)
- A "variant result" mechanism, that allows how an expression is evaluated to depend (at run time) on what use will be made of the result. Enables many later optimizations, including...
- Deferred evaluation, in which some operations aren't required to be done immediately, but only when the result is actually needed. This enables merging of operations, and computation in parallel threads.

The Variant Result Mechanism

A new technique introduced in pqR allows the caller of the internal "eval" function on an expression to request a *variant result*. The procedure doing the evaluation may ignore this, and operate as usual, but if willing, it can return this variant, which may take less time to compute.

Integer sequences: The implementation of "`for`" and of subscripting can ask that an integer sequence (eg, from ":") be returned as just the start and end points, without actually creating a sequence vector.

Example:

```
A <- matrix(data,1000,1000)
s <- numeric(900)
for (j in 1:1000)           # No 1000 element vector allocated
   s <- s + A[101:1000,j] # No 900 element sequence allocated
                           #  (Does allocate a 900 element vector
                           #   to hold data from a column of A)
```

The Variant Result Mechanism (Continued)

AND or OR of a vector: The `all` and `any` functions request that just the AND or the OR of their argument be returned. The relational operators, and some others such as `is.na`, obey this request, returning the AND or OR, sometimes without evaluating all elements of their operands.

Example:
```
if (!all(is.na(v))) ...  # may not look at all of v
```

Sum of a vector: The `sum` function asks for just the sum of its vector argument. Mathematical functions of one argument are willing.

Example:
```
f <- function (a,b) exp(a+b)
sum(f(u,v))  # No need to allocate space for exp(u+v)
```

Transpose of a matrix: The `%*%` operator says it's willing to receive the transpose of an operand. If it gets a transposed operand, it uses a routine that does the transpose implicitly.

Example:
```
t(A) %*% B  # Doesn't actually compute t(A)
```

Deferred Evaluation

The variant result mechanism is one way "task merging" is implemented in pqR. Other forms of task merging are implemented using a deferred evaluation mechanism, also used to implement "helper threads" that can do some computations in parallel.

Deferred evaluation is invisible to the user (except for speed) — it's not the same as R's "lazy evaluation" of function arguments.

Key idea: When evaluation of an expression is deferred, pqR records not its actual value, but rather *how to compute* that value from other values.

Structuring Computations as Tasks

A task in pqR is a numerical computation (no lists or strings, mostly), operating on inputs that may also be computed by a task.

The generality of tasks in pqR has been deliberately limited so that they can be scheduled efficiently. A task procedure has arguments as follows:

- A 64-bit operation code (which may include a length).
- Zero or one outputs (a numeric vector, matrix, or array).
- Zero, one, or two inputs.

When the evaluation of `u*v+1` is deferred, two tasks will be created, one for `u*v`, the other for `X+1`, where `X` represents the output of the first task.

The dependence of the input of the second task on the output of the first is known to the scheduler, so it won't run the second before the first.

How pqR Tolerates Pending Computations

Since pqR uses deferred evaluation, it must be able to handle values whose computation is pending, or that are inputs of pending computations.

Rewriting the entirety of the interpreter, plus thousands of user-written packages, is not an option. So how does pqR cope?

Outputs of tasks whose computation is pending are returned from procedures like "eval" only when the caller explicitly asks for them (eg, using the variant result mechanism). Otherwise, such procedures wait for the computation to finish. Of course, only code that knows what to do with pending values should ask to get them.

Inputs of tasks, which must not be changed until the task has completed, may appear anywhere, even in user-written code. But correct code checks NAMED before changing such a value. In pqR, the NAMED function waits for any tasks using the object to finish before returning.

Helper Threads

The original use of deferred evaluation in pqR was to support computation in "helper threads". Helper threads are meant to run in separate cores of a multicore processor, with the "master thread" in another core.

The main work of the interpreter is done only in the master thread, but numerical computations structured as tasks can run in helper threads. (Tasks can also be done in the master thread, when the result of a computation is needed and no helper is available.)

Example (assuming at least one helper thread is used):

```
a <- seq(0,1,length=1000000)
b <- seq(3,5,length=1000000)
x <- a+b; y <- a-b
v <- c (x, y) # a+b and a-b are computed in parallel
```

Pipelining

In general, when task B has as one of its inputs the output of task A, it won't be possible to run task B until task A has finished.

But many tasks perform element-by-element computations. In such cases, pqR can *pipeline* the output of task A to the input of task B, starting as soon as task A starts.

Consider, for example, the vector computation `v <- (a*b) / (c*d)`.

Without pipelining, the two element-by-element vector multiplies could be done in parallel, but the division could start only after both multiplies have finished.

With pipelining, all three tasks can start immediately, with the two multiply tasks pipelining their outputs to the division task.

Task Merging

A second use of deferred evaluation is to permit *task merging*.

As we've seen, some task merging can be done with the variant result mechanism, which has very low overhead. But using variant results to merge multiple diverse tasks would be cumbersome.

Instead, when a task procedure for an element-by-element operation is scheduled, pqR checks whether it has an input that is the same as its output, and that is also the output of a previously scheduled task. If so (and other requirements are met), it merges the two tasks into one. The previous task might itself be the result of a merge.

Example: All operations can be merged in `v <- exp(-v/2)`.

The merged task can compute the result in a single loop over elements of `v`, eliminating the need for memory stores and fetches of intermediate results.

Merged Task Procedures in pqR

Possible merged tasks in pqR are presently limited to sequences of certain operations with a single real vector as input and output, namely:

- many one-argument mathematical functions (eg, `exp`).
- addition, subtraction, multiplication, and division with one operand a vector and the other a scalar.
- raising elements of a vector to a scalar power.

At most three operations can be merged. This is because code sequences for all possible merged sequences (2744 of them) are precompiled and included in the interpreter.

Timing Comparison of R-3.2.5 versus pqR-2015-09-14

Intel Xeon X5680, 3.33 GHz, 6 cores, both compiled with gcc 4.9.2.

```
> a <- rnorm(10000); b <- rnorm(10000)
> system.time (for (i in 1:100000) x <- a %*% b)
   user  system elapsed       user  system elapsed
 10.564   0.008  10.607      0.936   0.000   0.936

> system.time (for (i in 1:100000) x <- runif(1000))
   user  system elapsed       user  system elapsed
  3.576   0.000   3.591      1.932   0.000   1.938

> L <- list (a="a", b="b", c="c", abc=1, def=2, xyz=3)
> system.time (for (i in 1:10000000) x <- L$xyz + L$abc)
   user  system elapsed       user  system elapsed
  7.953   0.016   7.994      3.584   0.000   3.598

> a <- 1:10000
> system.time (for (i in 1:100000) x <- any(a^2>10))
   user  system elapsed       user  system elapsed
 10.321   1.156  11.511       2.18    1.28    3.47
> system.time (for (i in 1:100000) x <- any(a^2>1e100))
   user  system elapsed       user  system elapsed
 12.372   0.896  13.309      3.472   1.256   4.740
```

R-3.2.5 pqR-2015-09-14

```
> v <- seq(0.1,by=0.1,length=100000)
> system.time (for (i in 1:10000)
+                x <- sum(sqrt(v[200:80000])))
    user  system elapsed       user  system elapsed
  14.377   0.156  14.580      8.745   1.452  10.229

> M <- matrix(1:160000,400,400)
> system.time (for (i in 1:10000)
+                X <- M[1:200,1:200])
    user  system elapsed       user  system elapsed
   1.796   0.000   1.803      0.280   0.028   0.310

> a <- seq(1,2,length=10000)
> r <- list(x=0)
> system.time (for (i in 1:10000) r$x <- (3*a+1)/5)
    user  system elapsed       user  system elapsed
   0.912   0.004   0.917      0.724   0.004   0.728
> system.time (for (i in 1:10000)
+                r$x <- sin((exp(a)+exp(-a))/a))
    user  system elapsed       user  system elapsed
   9.673   0.000   9.705      8.464   0.012   8.507
```

R-3.2.5 pqR-2015-09-14

Benefit of Helper Threads

The last test can be used to show the benefits of helper threads.

R-3.2.5:

```
> system.time (for (i in 1:10000) r$x <- sin((exp(a)+exp(-a))/a))
   user  system elapsed
  9.673   0.000   9.705
```

pqR-2015-09-14 with no helper threads:

```
> system.time (for (i in 1:10000) r$x <- sin((exp(a)+exp(-a))/a))
   user  system elapsed
  8.464   0.012   8.507
```

pqR-2015-09-14 with 5 helper threads (plus master thread):

```
> system.time (for (i in 1:10000) r$x <- sin((exp(a)+exp(-a))/a))
   user  system elapsed
 22.854   0.012   3.841
```

Future Performance Improvements in pqR

- More operations done as tasks that can be parallelized or merged with other operations — eg, extracting subsets, as in `a <- vec[100:200]^2`.
- Allow list elements to have their evaluation deferred, increasing the potential for task merging and parallel evaluation.
- Better integration of task merging with parallelization, allowing merging with operations that have already started to execute in a helper thread.
- Automatic parallelization of single vector operations, when the vectors are sufficiently long.
- Automatic tuning of when to defer / parallelize operations, taking account the overhead of doing so.
- Rewriting the garbage collector, to reduce the size of objects.

Language Extensions Planned for pqR

Plan for Language Extensions

The latest version of pqR has a completely re-written parser. This has performance benefits, fixes a number of bugs, and cleans up the code a lot.

It also makes it easier to implement language extensions.

The plan for pqR is to extend R in ways that:

- fix design errors in R, or
- make writing R code more convenient, or
- provide substantial new facilities, while
- being completely or largely compatible with existing R code.

  Some slight incompatibilities can mostly be handled by allowing the option of disabling potentially problematic pqR extensions, which can be set automatically for code in packages not designed for pqR.

Some syntactic sugar (or maybe more than that...?)

- For any non-S4 object, `x`, make:

  `x@fred` equivalent to `attr(x,"fred")`

  `x@fred <- v` equivalent to `attr(x,"fred") <- v`

- For any matrix, `X`, make:

  `X$fred` equivalent to `X[,"fred"]`

  Now code designed for data frames can be used unchanged on matrices.

- Extend `for` to allow `along` instead of `in`:

  `for (i along vec)` equivalent to `for (i in seq_along(vec))`

  `for (i,j along M)` equivalent to

  ```
  for (j in seq_len(ncol(M))) for (i in seq_len(nrow(M)))
  ```

A New Sequence Operator that Operates Correctly

Problem: Using `i:j` to create an increasing sequence does not produce a zero-length sequence when `j` is less than `i`. This is very annoying, and leads to buggy code. Using `seq_len` is clumsy and not sufficiently general.

Another problem: `1:n-1` does not start at `1`.

Solution: A new operator, which produces only increasing sequences, including zero-length ones, and which has lower precedence than the arithmetic operators.

Question: What should be the name of the new operator?

It's not a trivial question. We need to make ":" obsolete, but retain it for compatibility, so we can't redefine it. But the new operator won't take over from ":" if its name is ugly and/or hard to type.

Examples using possibilities I've considered but don't like:

```
for (i in 1 %:% n-1) A[i, i %:% i+1] <- 0
for (i in 1 :> n-1) A[i, i :> i+1] <- 0
```

Solution: Call the New Sequence Operator "..."

Examples of its use:

```
for (i in 1..n-1) A[i, i..i+1] <- 0

v[1..n] <- A[1..n,i]

if (any(v<i..j)) stop(...)
```

But... `i..j` is a valid symbol!

Yes. I disallow symbols with consecutive dots (except at the beginning of the symbol, so "`...`" and "`..1`" are still legal).

Does anyone use symbols with "`..`" in the middle? I hope not.

It would be good (anyway) to encourage use of underscores rather than dots in symbols (except for S3 method names). I think the expression `i..max_pens` looks better than `i..max.pens`, though the latter remains unambiguous.

Stopping Inadvertent Dimension Dropping

Problem: We want to create a sub-array of `A` with all its columns, but only those rows whose indexes are in `v`. We try to do that with `A[v,]`.

It usually works, but we get a vector rather than a matrix if either `v` has length one or `A` has only one column. So there's lots of buggy code. Adding `drop=FALSE` everywhere works, but is very tedious and unreadable.

Start of a solution:

First, define "`_`" to be a special object equivalent to a missing argument, so it selects all of a dimension — but without dropping it, even if the dimension is one. Writing "`_`" is also clearer than writing nothing.

Second, don't drop a dimension if the index is a 1D array, even if it is of length one. This might break some existing code, but probably very little.

Result: Now `A[array(v),_]` always produces a matrix.

Make the New Sequence Operator Produce a 1D Array

Many of the vectors used to index arrays are produced by a sequence operator. We can define the new sequence operator to produce a 1D array, so we don't have to use `array`.

Now, `A[1..n,_]` produces a matrix with one row when `n` is one, and a matrix with zero rows when `n` is zero.

Similarly, `A3[1..n,1,1..m]` delivers a 2D matrix even when `n` and/or `m` is zero or one. Note that adding `drop=FALSE` would not solve the problem here, since it would always produce a 3D array.

An Unfortunately Impossibility: Zero-Length Vectors Can't Contain Negative Elements

Problem: If `ix` is a vector of positive integers, `v[-ix]` gives a vector with all the elements of `v` except those in `ix`.

Well, almost. Unfortunately, it doesn't work when `ix` is of length zero!

Solution: Define a function `except(ix)` that returns `ix` with some suitable attribute attached that signifies exclusion rather than inclusion.

Now `v[except(ix)]` works correctly when `ix` happens to have length zero. Also, it can now work with indexes that are names.

It's maybe clearer too. Plus, we can now find bugs more easily, if we make zero and negative numbers in `ix` illegal.

Flags for Functions

Several needs can be addressed by allowing specification of certain "flags" on function definitions, on the formal arguments for functions, or on the actual arguments when the function is called.

Example syntax:

```
f <- function (top=1, bottom \!lazy) \closed \exact { ... }
f (bottom = g() \lazy, middle = h() \ignorable)
```

Possible function flags:

`closed`	Variable lookups outside the function go directly to the base environment
`pure`	Function is not allowed to have side effects (including via functions it calls)
other	Default value for argument flags

Defining a closed function prevents the error of inadvertently referring to a function argument you forgot to define, or are using the wrong name for. This may appear to work because you have a global variable of the same name.

Argument Flags

Possible formal argument flags:

`!lazy`	Lazy evaluation is not done for this argument
`exact`	Argument name must match exactly
`integer`	Must be convertable without loss to type integer
`real`	Must be convertable without loss to type real
`character`	Must be convertable without loss to type character
`scalar`	Must be a vector of length one
`vector`	Must be a vector with no dimensions or one dimension
`matrix`	Must be a matrix
`positive`	All elements must be greater than zero
`nonnegative`	All elements must be greater than or equal to zero

Possible actual argument flags:

`ignorable`	Argument can be ignored if it doesn't exist
other	Overrides formal argument flag

More Possible Language Extensions

- More convenient ways to create lists and objects with attributes:

```
L $$ a=1 $$ b=2   # equivalent to c(L,list(a=1,b=2))
$$ ab $$ cd $$ ef # equivalent to list(ab=ab,cd=cd,ef=ef)


1:6 @@ dim=c(2,3) @@ class="fred"  # tack on attributes
```

- A way to return more than one item without putting them all in a list. Would allow what is returned to be extended without invalidating existing calls, just as argument defaults allow what is passed to be extended.

```
f <- function (x) { grad <- function () ...
                    hess <- function () ...
                    value <- ...
                    value, gradient=grad(), hessian=hess() }
u <- f(a)                    # the gradient and hessian are
v, gradient=g <- ... f(b)    # evaluated lazily, as needed
```

- Argument passing using the "call-by-name" mechanism of Algol 60.

Some other plans for pqR

- Exactly-rounded sums and means. Exactly-rounded computation of means should be faster than the present method, and exactly-rounded sums only slightly slower. I have a blog post and an arxiv paper on this.

- Compact representations of objects. Represent logicals in one byte or in two bits, represent diagonal matrices by just the diagonal, etc.
  Do it automatically and invisibly. I have a blog post on this too, and an arxiv paper on compact floating-point representations.

- Support for automatic differentiation, via general support in the interpreter for tracking how a value was computed (also useful for debugging and other purposes).