Speed Improvements in pqR:
Current Status and Future Plans

Radford M. Neal, University of Toronto

Dept. of Statistical Sciences and Dept. of Computer Science

`http://www.cs.utoronto.ca/∼radford`

`http://pqR-project.org`

History and Current Status of 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 4 years later).

After further work, I released the first version of pqR, a "pretty quick" R, in June 2013. The current version is pqR-2014-06-19.

Some improvements from pqR have been put into R-3.1.0, but most not.

Detailed Code Improvements in pqR

Some of the “low hanging fruit” for speeding up R takes the form of local rewrites of code, without any global changes to the design. Examples of operations that have been sped up in this way are

- Subsetting of vectors and matrices (eg, `M[1:100,100:2000]`).
- Finding the transpose of a matrix (the “`t`” function).
- Generation of random numbers (eg, avoid copying the random seed, which for the default generator consists of 625 integers, on every call).
- The “`$`” operator for accessing list elements.
- Matching of arguments passed to functions with their names within the function.
- Many others...

Limited Redesign

Other speedups in pqR come from redesigning the interpreter in limited ways that don't have global implications. Examples are:

- Providing a "fast" interface to simple primitive functions/operators, when there are no complicating factors such as named arguments.

  This is a **big** win, partly because the "slow" interface requires allocation of a storage cell for every argument, which will later have to be recovered by the garbage collector.

- A way of quickly skipping to the definition of a standard operator (eg, "`if`" or "`+`") when it hasn't been redefined. This has been incorporated in R-3.1.0.

Moving Towards Real Reference Counting

The R Core implementations sort of do reference counting, with a NAMED field that is limited to representing a count of 0, 1, or 2-or-more. Assigning or passing an object just changes its NAMED field. Copying is done when an object with NAMED greater than 1 needs to be changed.

For example:

```
A <- matrix(0,1000,1000)  # create a matrix, NAMED will be 1
A[1,1] <- 7               # no copy done
B <- A                    # still no copy, just changes NAMED
B[2,2] <- 8               # a copy has to be made, since
                          # NAMED for B (hence also A) is 2
A[3,3] <- 9               # unfortunately, makes another copy!
```

In pqR, a 3-bit NAMEDCNT field allows for counts up to 7, and some attempt is made to decrement counts - eg, avoiding the extra copy above.

R-3.1.0 has an experimental reference counting scheme more general than pqR's (but with only a 2-bit field at present), which I may adopt for pqR.

The Variant Result Mechanism

A new technique introduced in pqR allows the caller of "eval" for 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.

Renjin and Riposte also do deferred evaluation, in a rather general way. In pqR, only certain numerical operations can be deferred — those whose computation has been implemented as a pqR “task procedure”.

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

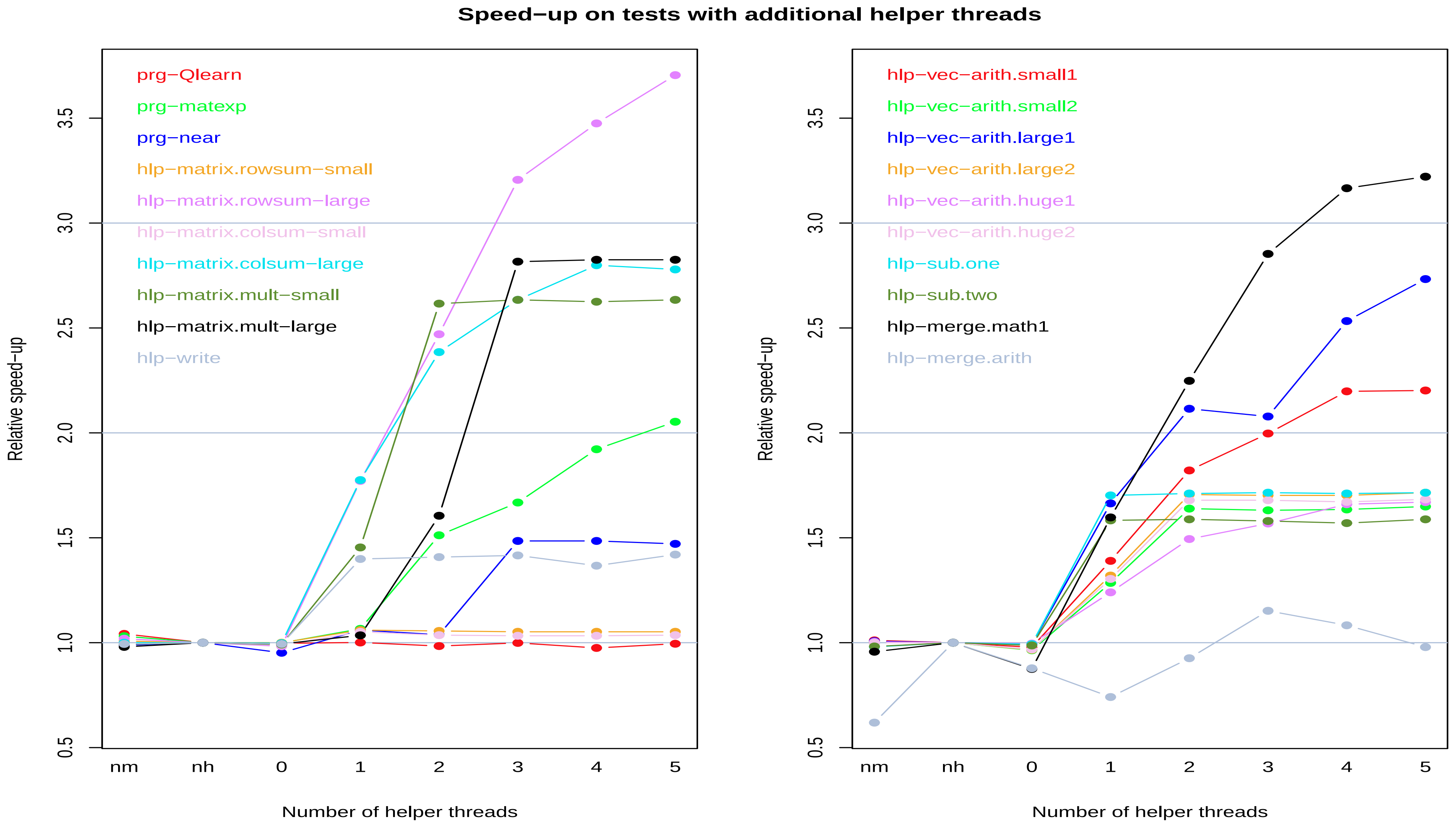
Possible merged tasks in pqR are presently limited to sequences of certain operations with a single real vector as input and output, namely:

- 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.

Renjin and Riposte use more general schemes, and generate compiled code on-the-fly. It will be interesting to see how much of the benefit of task merging can be obtained with the more rudimentary scheme in pqR.

Speedup Using Helper Threads on Simple Test Programs



Some Plans for Further pqR Speed Improvements

- More detailed code optimization. Still lots to be done!
- Improvements to the garbage collector.
- Write more operations as tasks (eg, subsetting, such as `v[101:200]`), which can then be done in helper threads, or merged with other operations (eg, `u <- 2*v[101:200]+1` could be done in one loop).
- Use the variant result mechanism to make `v <- 2*v+1` update `v` in place (if the two instances of `v` actually refer to the same variable!).
- Allow some calls of user-defined procedures (with `.C` or `.Fortran`) to be done in helper threads.
- Allow objects whose computation is pending to appear in more places — eg, as list elements. More deferral $\rightarrow$ more opportunities for merging and parallel evaluation.

More Sophisticated Helper Threads

Use of helper threads could also be improved:

- More than one processor core could be used to do a single task.

  This might be done only when an idle helper thread is available, and, for a task with pipelined input, only when the task is falling behind in processing its input.

- A task could stop when a new task it can merge with is scheduled.

  One would want the part of the processing done so far to be updated with the second operation, and then the rest of the input done with a merged procedure.

Both of these seem possible to do, but a bit tricky!

The Need for Automatic Tuning

There is some overhead to scheduling tasks — so small ones should just be done immediately by the master thread.

Merging tasks is sometimes better than doing them in parallel with pipelining — and sometimes not.

A system's cache size and processor core hierarchy affect what is fastest. So may the compiler used.

Currently, pqR has uses some hand-coded thresholds for deciding whether to merge and/or allow use of a helper thread. But these guesses really need to be replaced by some automatic tuning mechanism.

Tests Showing Performance Varying with Processor Type

Tests with use of helper thread disabled vs. one helper thread, with vector `a` of length 10^3 or 10^5, giving times in seconds for each of three repetitions of a loop doing 10^8 total evaluations.

	X5680 (6 cores, 3.33 GHz)			Core 2 Duo (2 cores, 1.4 GHz)			2 × E5262 (4 cores, 2.8 GHz)			2 × Xeon (1 core, 1.7 GHz)		
	no hlp	1 hlp	ratio	no hlp	1 hlp	ratio	no hlp	1 hlp	ratio	no hlp	1 hlp	ratio
`(a+1.2)*a`	0.327	0.318	1.03	1.067	1.001	1.07	0.587	0.955	0.61	1.865	2.459	0.76
length 10^3	0.332	0.318	1.04	1.074	0.997	1.08	0.599	0.944	0.71	1.838	2.305	0.80
	0.332	0.316	1.05	1.074	0.998	1.08	0.604	0.704	0.86	1.871	2.272	0.82
	0.365	0.281	1.30	1.460	1.289	1.13	0.673	0.813	0.83	3.194	3.179	1.00
length 10^5	0.384	0.277	1.39	1.472	1.261	1.17	0.664	1.276	0.52	3.217	3.157	1.02
	0.360	0.281	1.28	1.448	1.284	1.13	0.647	1.292	0.50	3.186	3.177	1.00
`(a+1.3)/a`	0.896	0.866	1.03	2.337	2.924	0.80	1.252	2.272	0.55	3.917	3.749	1.04
length 10^3	0.897	0.859	1.05	2.313	2.825	0.82	1.254	2.392	0.52	3.915	3.747	1.04
	0.896	0.867	1.03	2.336	2.814	0.83	1.249	2.326	0.54	3.918	3.747	1.05
	0.913	0.813	1.12	2.665	2.357	1.13	1.250	1.078	1.16	4.427	3.477	1.27
length 10^5	0.918	0.806	1.14	2.682	2.290	1.17	1.266	1.299	0.97	4.408	3.504	1.26
	0.915	0.807	1.13	2.671	2.317	1.15	1.250	1.325	0.94	4.470	3.479	1.28
`sin(exp(a))`	5.057	2.768	1.83	9.499	6.711	1.42	4.451	4.691	0.95	27.962	15.983	1.75
length 10^3	5.055	2.763	1.83	9.520	6.588	1.45	4.449	4.882	0.91	27.964	16.317	1.71
	5.055	2.759	1.83	9.516	6.563	1.45	4.447	4.333	1.03	27.962	15.989	1.75
	5.075	2.610	1.94	9.906	5.854	1.69	4.503	3.030	1.49	28.423	15.094	1.88
length 10^5	5.078	2.621	1.94	9.904	5.848	1.69	4.519	3.666	1.23	28.374	15.111	1.88
	5.083	2.610	1.95	9.933	5.947	1.67	4.519	3.647	1.24	28.406	15.308	1.86

Conclusions

Advantages of pqR:

- Based on the R Core implementation — so has a high degree of compatibility (though currently with R-2.15.0).
- Many, many detailed speed improvements.
- Allows (some) numerical compuations to be done in parallel, with each other, and with interpreted operations.
- Implements dynamic forms of task merging and specialization.
- Not specific to one architecture — eg, works on a Raspberry Pi (ARM).
- Code for managing helper threads and task merging is a separate module — could be used elsewhere (eg, Octave?).

Disadvantages of pqR:

- Currently, many optimizations are not enabled in byte-compiled code.
- Doesn't take advantage of work done on implementing other languages (eg, JVM).