Speeding up R with Multithreading, Task Merging, and Other Techniques

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`

Talk at University of Guelph, Dept. of Mathematics & Statistics, 29 November 2013

Part I: Background on R and on Implementations of R

The R Language and Its Applications

- R is based on S — like S, it is designed for interactive use, as well as programming.
- Comes with many, many statistical packages.
- Some special language features support statistical analysis (eg, NA values, names for rows and columns of matrices).
- Supports operations on vectors and matrices — eg, `u+v` could add two long vectors, `u` and `v`.
- Used by statisticians in diverse environments:
  - Statistical research
  - Development of statistical software
  - Data analysis
  - Teaching / learning

The R Core Implemention of R

R was initially written by Robert Gentleman and Ross Ihaka.

This implementation was based on a Lisp interpreter, traces of which may still be seen in the R langage and implementation (eg, the occassional appearance of “pairlists”).

The R Core Team have continued development of this implementation. Current active members include Luke Tierney, Brian Ripley, and Duncan Murdoch.

R-1.0.0 was released February 2000; R-3.0.2 is the latest release.

This R Core implementation is distributed under the Gnu General Public License (GPL), which allows anyone to distribute modified versions.

How I Came to be Working on 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).

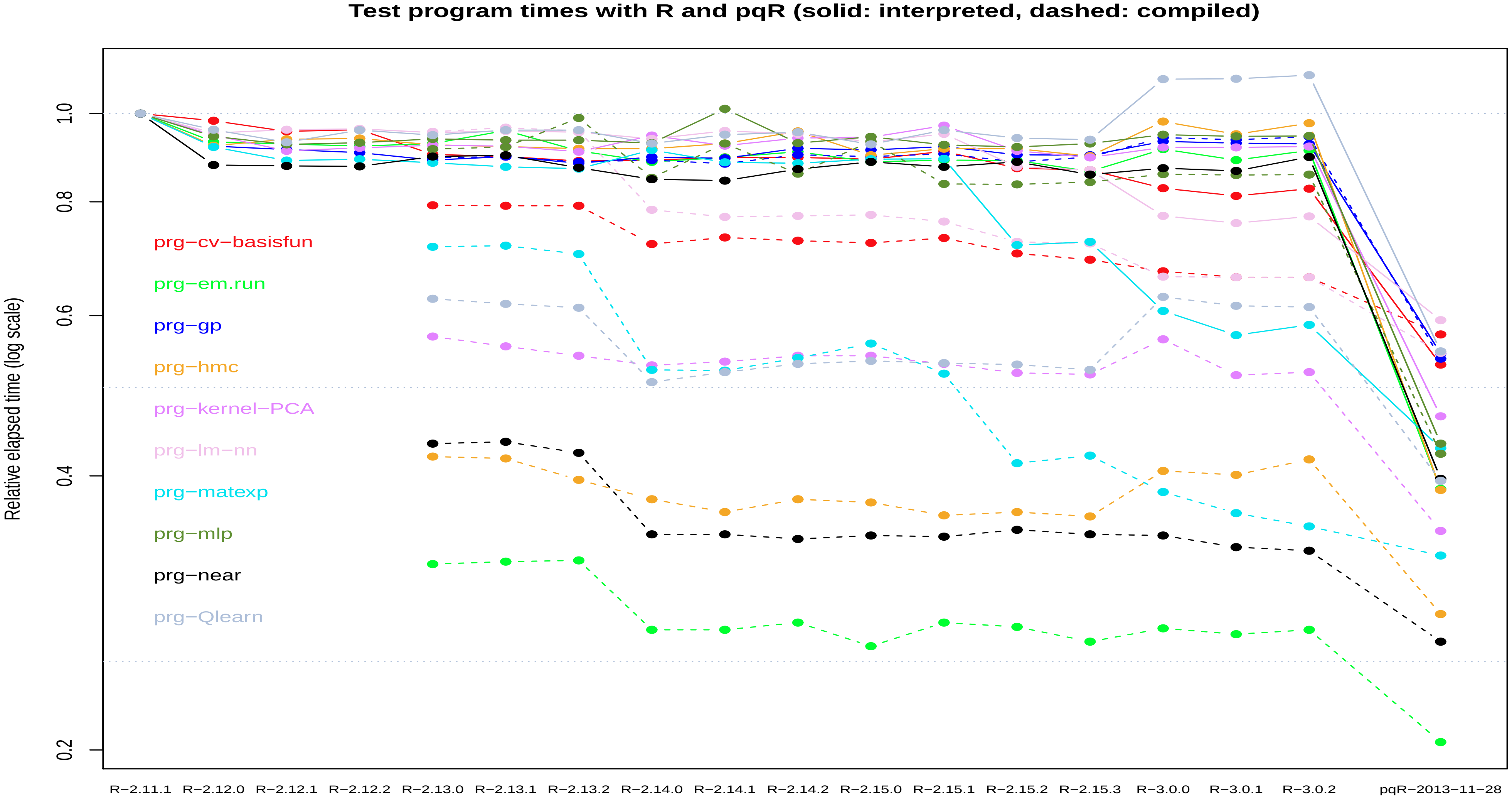
I realized that there was much "low hanging fruit" in the R interpreter, and made patches to R-2.11.1 which sped up parentheses, squaring, and several other operations, including reducing general overhead.

Variants of a few of my patches were incorporated into R-2.12.0, but the R Core Team was uninterested in most of them — eg, a small patch to speed up matrix-vector multiplies by about a factor of five. It was clear that speeding up R would require creation of a separate implementation.

I released the first version of pqR — a "pretty quick" R — in June 2013.

Speed of the R Core Implementation and of pqR

The speed of R Core and pqR implementations over time (for some small programs mostly dominated by interpretive overhead):



Challenges in Implementing R

Though there's much low-hanging fruit in the R Core implementation, there are also some real difficulties in making a faster R:

- Variables can hold values of any type:

  ```
  f <- function (x) 2*x
  ```

  The multiplication in `f` could be of a value for `x` that is integer, real, complex, or any user-defined class that implements the `*` operator.

- Variables can be accessed by non-obvious links to their environment:

  ```
  f <- function (x) { y <- g(x); 2*y }
  ```

  There is no guarantee that the variable `y` is unused after `f` returns, since `g` may have somehow obtained a reference to `f`'s environment.

- Standard facilities can be redefined:

  ```
  f <- function (x) { ‘*‘ <- function (a,b) b+10; 2*x }
  ```

  The call `f(7)` evaluates to 17.

Other New Implementations of R

Several other projects are writing new R interpreters “from scratch”. Two of these projects share with pqR the use of deferred evaluation to support multithreading and task merging:

> **Renjin** (at `renjin.org`) is based on the Java Virtual Machine. One of its aims is to allow access to (possibly big) data stored anywhere, in the same way as it is accessed when in memory. Its initial version was launched July 2013.
>
> **Riposte** (at `https://github.com/jtalbot/riposte`) has as one main aim to make vector calculations in R as fast as in efficient C code. It aims to launch in July 2014.

Both Renjin and Riposte are more ambitious than pqR, but consequently require more implementation effort, and may possibly have problems achieving compatibility with R Core’s implementation.

Part II: Speed Improvements in pqR

Detailed Code Improvements

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

- Matrix multiplication (both by the routines supplied with R and when calling external optimized BLAS routines).
- 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, the "if" or "+" operator) when it hasn't been redefined.

Avoiding Unnecessary Copying

R gives the illusion that assignment and argument passing *copy* the value assigned or passed, so it does not change when the original object changes.

Examples:

```
a <- c(0,10,100)
b <- a
a[1] <- 99;
print(b[1])  # prints 0, not 99

f <- function (x) { x[1] <- x[1]+1; sum(x^2) }
a <- c(0,10,100)
b <- f(a)
print(a[1])  # prints 0, not 1
```

If R always *actually* did these copies, it would be horribly inefficient — eg, a huge matrix passed as an argument would be copied even if the function just looks at a few elements of this matrix without changing any.

Avoiding Unnecessary Copying (Continued)

So the R Core implementation of R doesn't always copy objects when they are assigned or passed as arguments.

Each object has a "NAMED" field that records whether the number of "names" referring to it is 0, 1, or 2 or more. Assigning or passing an object just changes its NAMED field. An actual copy is done when an object with NAMED greater than 0 (or 1 in some cases) 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!
```

Problems with NAMED in the R Core Implementation

The NAMED mechanism avoids copies only in some situations, beause the R Core implemenation makes no attempt to ever *decrease* NAMED. Once NAMED reaches 2, it stays 2 forever.

One consequence is seen in the previous slide. Another is that an object passed as an argument of a function will be copied when next modified:

```
f <- function (M) M[1,1]+M[100,100]
A <- matrix(data,100,100)
print(f(A))
A[1,2] <- 0   # A is copied before A[1,2] is changed
```

The NAMED mechanism is also not adequately documented — it actually needs to count not just "names" but also references from list elements, and also from attributes (?), pairlists (?), ... This lack of clarity has probably contributed to a number of NAMED-related bugs recently found in the R Core implementation (some as a result of my work on pqR).

How pqR is Addressing the NAMED Problems

In pqR, the NAMED field is replaced by a NAMEDCNT field that can hold a count of up to 7. Macros are provided that maintain backwards compatibility with packages that refer to the old NAMED field.

In some situations (eg, the first previous example, where both A and B are modified), pqR can decrement NAMEDCNT, and thereby avoid unnecessary copies.

However, in other situations (eg, the second example, where A is passed as an argument) keeping NAMEDCNT exact is currently too difficult.

The documentation for pqR also makes a start at clarifying exactly what the meaning of NAMEDCNT/NAMED is.

There is more work to be done here, however.

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, 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 OR of their argument be returned. The relational operators, and some others such as “is.na”, obey this request, and return the AND or OR without necessarily 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 asks for the transpose of its operands. If it gets a transposed operand, it uses a routine that operates directly on the transpose.

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”.

Deferred evaluation is invisible to the user (except from 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 immediately, 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, these procedures wait for the computation to finish before returning the value. Of course, only code that knows what to do with such 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 before such code (if correct) changes such a value, it will first check NAMED/NAMEDCNT. The NAMED/NAMEDCNT functions are now written to wait for the 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 kinds of 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, a check is made for whether it has an input that is the same as its output and is also the output of a previously scheduled task. If so (and if other requirements are met), the two tasks can be merged into one. The previous task might itself be the result of merging two tasks.

The merged task can compute the result of all merged operations in a single loop over elements, eliminating the need to store and fetch intermediate results to and from main memory.

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.

Some Examples Using Helper Threads and Task Merging

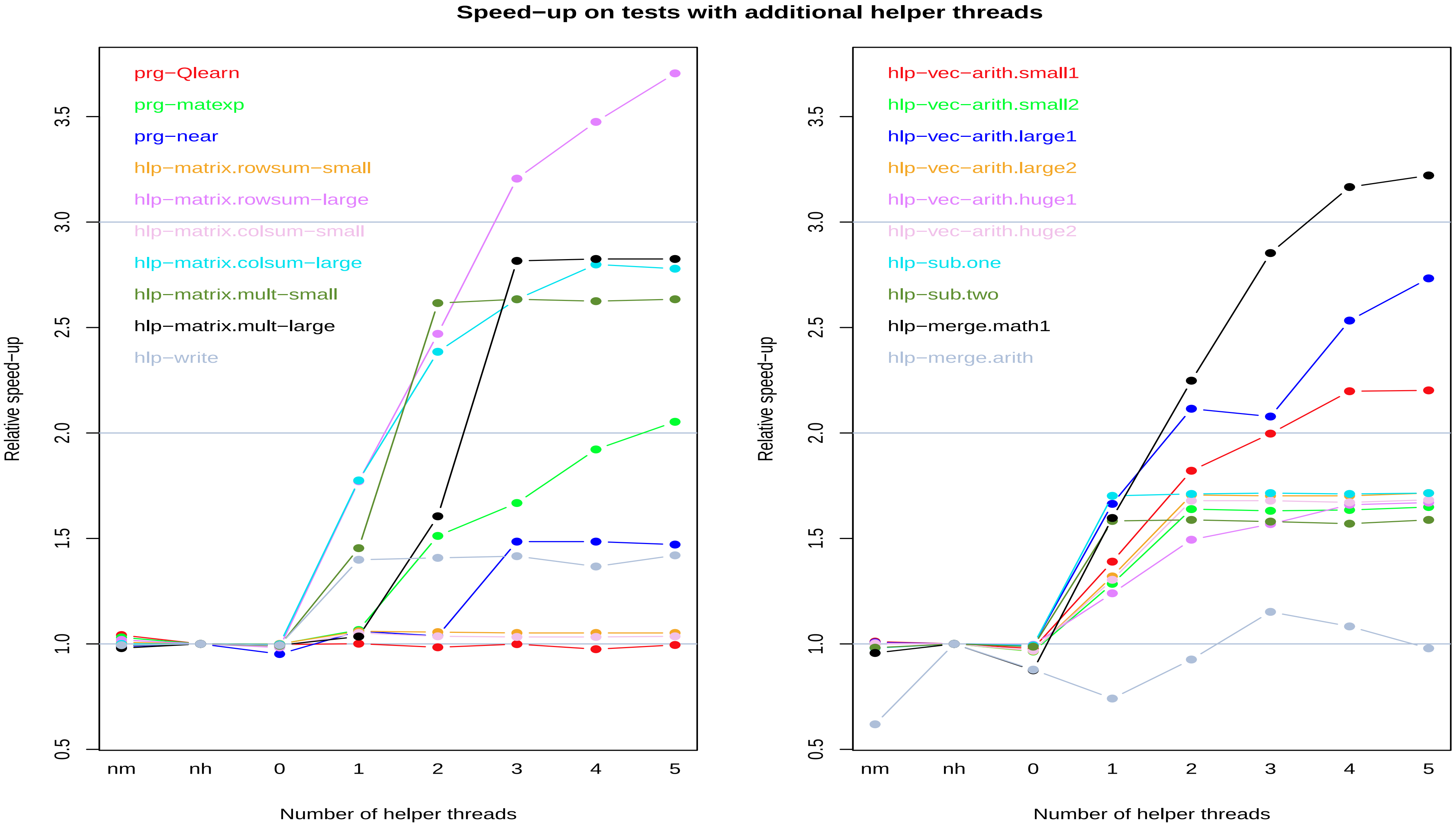
```
n <- 1000000; rep <- 100; a <- seq(1,2,length=n); b <- seq(2,4,length=n)

s <- 0; system.time( for (i in 1:rep) { v <- a/b + b/a; s <- s + v[n] } )
s <- 0; system.time( for (i in 1:rep) { v <- exp(a)/b;  s <- s + v[n] } )
s <- 0; system.time( for (i in 1:rep) { v <- 3.1*a+4.2; s <- s + v[n] } )
```

Times on a 6-core Intel Xeon X5680, at 3.33 GHz:

R-3.0.2	pqR (no merge)	pqR (0 helpers)	pqR (1 helper)	pqR (2 helpers)
2.160	1.803	1.794	1.067	0.951
3.306	2.737	2.740	2.327	2.311
0.540	0.366	0.237	0.257	0.270

Speedup Using Helper Threads on Simple Test Programs



Part IV: Future Directions for pqR

More Speed Improvements

- More detailed code improvements. Still lots to be done!
- Write more operations as tasks, that can be done in helper threads.
- 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.
- Extend task merging to include vector subsetting and vector sum, difference, and product. Example: `u+2*v[100:199]`, with `u` a vector.
- Continue to address the NAMED problems in R, to avoid unnecessary copies (and bugs).

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!

Language Extensions

Fixing some design flaws in S/R. Two related problems:

- `for (i in 1:n) ...` does the loop twice when `n` is zero!
- After `A <- matrix(data,n,m)`, the expression `A[a:b,c:d]` gives a vector, not a matrix, if `a==b` or `c==d`.

Solution? Define an operator like ":" (maybe ".."?) that produces only an ascending sequence (maybe length 0) with a "dim" attribute indicating it should be treated as a vector, not a scalar, when used as a subscript.

Allowing procedures to update objects. Perhaps like this:

```
f <- function (@a,i) a[i] <- a[i-1] <- a[i+1] <- 0
L <- list (u = numeric(100), v = numeric(200))
f(@L$v,20)  # set L$v[19], L$v[20], and L$v[21] to zero
```

Semantics could be the same as the "call by name" mechanism of Algol 60.