|  |  |
| --- | --- |
|  | The [latest version of pqR](https://radfordneal.wordpress.com/2018/11/25/new-version-of-pqr-with-major-speed-improvements/) and the version before as well use a new garbage collector, and new memory layouts for R objects, which both reduce memory usage and considerably speed up garbage collection. |

Here, I’ll give an overview of how the new scheme works, and present some performance comparisons with R-3.5.1.

**Allocating memory in segments**

Objects in the new scheme are stored in “segments” — many objects per segment for small objects, just one per segment for big objects. This allows objects to be identified by a segment identifier and an offset within a segment (measured in “chunks”, currently 16 bytes in size), which together fit in 32 bits regardless of the size of a machine address.

It’s possible to configure pqR to use such 32-bit “compressed pointers” for all references, which reduces memory usage considerably on machines with 64-bit addresses, though at a cost of up to a 40% slowdown for scalar code dominated by interpretive overhead. (There are also compatibility issues with Rcpp and Rstudio when compressed pointers are used).  The default is still to use machine addresses for references in R objects, but compressed pointers are always used internally by the garbage collector.

The new scheme reduces the space occupied by an R object even if references in the object do not use compressed pointers.  The garbage collector needs to keep track of several sets of objects — for example, newly-allocated objects versus objects that were retained after the previous garbage collection.  For this purpose, the old R Core garbage collector requires that every object contain two pointers used to implement such sets as doubly-linked lists.  The new pqR garbage collector instead represents these sets much more compactly as bit vectors.  If pqR is configured so object references are not done with compressed pointers, each object needs to store a compressed pointer to itself to allow the garbage collector to access these bit vectors, but that takes only 4 bytes, much less than the 16 bytes needed for two 64-bit pointers.

**Memory reference locality**

A full garbage collection requires that all accessible objects be scanned, and marked for retention. This can potentially result in accesses scattered over large areas of memory, many of which would not be to cache. On modern computers, an access to a memory location not in a cache can be hundreds of times slower than an in-cache reference.

This problem is reduced by the more compact layout of objects in pqR (considerably more compact if compressed pointers are used, and still somewhat more compact if not), since if the total memory occupied is smaller, a larger fraction of it will fit in cache. Locality of reference is also important, since an accesses to a location near to one accessed recently will likely go to a cache.

The use by pqR of bit vectors to represent membership in sets, including the set of objects that have been marked for retention, helps with locality. These bit vectors are stored in 64-byte structures associated with segments, allocated in blocks, which should often result in good locality of access. In contrast, with the old R Core garbage collector these operations involve accessing and writing to a “mark” bit in the object header and accessing and modifying the pointers in an object used for the doubly-linked lists. These accesses will be scattered over the whole area of memory used to hold objects.

**Speed comparisons**

It’s difficult to conduct meaningful speed comparisons of the garbage collector alone between pqR and R Core implementations, since they differ not just in their garbage collectors, but also in how many objects they allocate, and how many objects exist that may need to be scanned during garbage collection.

Regarding the last point, the R Core implementation is at a disadvantage because in its recommended configuration all the base R functions will be byte-compiled, increasing the number of objects that need to be scanned in a full garbage collection, whereas byte-compilation is not recommended for pqR. This is not a difference in the garbage collectors themselves, however.

But one can get some insight by looking at the performance of R code for which garbage collection speed might be expected to be an issue. In two tests I show below, garbage collection is more significant in the second than in the first, because in the second test many objects are allocated, retained for some time, but finally recovered.

Here are the two tests run (separately) with pqR-2018-11-18:

> a<-c(3,4,1); r <- rep(list(0),100)

> system.time(for (i in 1:100000) for (j in 1:100)

+ r[[j]] <- list(x1=a+1,x2=a-1,x3=a+2,x4=a-2))

user system elapsed

5.993 0.000 5.993

> a<-c(3,4,1); r <- rep(list(0),100000)

> system.time(for (i in 1:100) for (j in 1:100000)

+ r[[j]] <- list(x1=a+1,x2=a-1,x3=a+2,x4=a-2))

user system elapsed

8.217 0.041 8.257

And here are the same tests run with R-3.5.1 (with the JIT enabled):

> a<-c(3,4,1); r <- rep(list(0),100)

> system.time(for (i in 1:100000) for (j in 1:100)

+ r[[j]] <- list(x1=a+1,x2=a-1,x3=a+2,x4=a-2))

user system elapsed

5.238 0.008 5.246

> a<-c(3,4,1); r <- rep(list(0),100000)

> system.time(for (i in 1:100) for (j in 1:100000)

+ r[[j]] <- list(x1=a+1,x2=a-1,x3=a+2,x4=a-2))

user system elapsed

14.098 0.031 14.129

One can see that R-3.5.1 is a bit faster than pqR-2018-11-18 for the first test, but much slower for the second test. The difference seems to be due to pqR’s faster garbage collector. For the first test, the Linux “perf record” command reveals that both implementations spend less than 5% of their time in the garbage collector (much less than 5% for pqR). For the second test, about 57% of the compute time for R-3.5.1 is spent in the garbage collector, whereas pqR-2018-11-18 spends about 12% of its time in the garbage collector during this test. The faster garbage collection seen here for pqR is presumably due to the factors such as locality discussed above.

The R Core garbage collector is also slower for a more specific reason, involving handling of character strings. Both pqR and R Core garbage collectors are of the “generational” sort, in which most garbage collections only attempt to recover unused objects that were recently allocated, and consequently do not have to scan old objects that were allocated long ago (they are recovered if no longer used only in the infrequent full collections). But in the R Core implementation, even these partial garbage collections scan *all* character strings. Consequently, as more strings are kept around, all operations that allocate memory (and hence may trigger garbage collection) become slower.

Here’s an illustration. First, some times with pqR-2018-11-18:

> a<-seq(0,1,length=100); n <- 1000000

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

0.477 0.004 0.480

> x<-paste("a",1:1000000,"a")

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

0.869 0.004 0.873

> y<-paste("b",1:1000000,"b")

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

0.899 0.000 0.898

> z<-paste("c",1:1000000,"c")

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

0.940 0.003 0.943

And here are the times with R-3.5.1:

> a<-seq(0,1,length=100); n <- 1000000

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

0.504 0.008 0.512

> x<-paste("a",1:1000000,"a")

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

1.975 0.000 1.975

> y<-paste("b",1:1000000,"b")

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

2.857 0.005 2.861

> z<-paste("c",1:1000000,"c")

> system.time(for (i in 1:n) r <- list(x=a+1,y=a-1))

user system elapsed

4.073 0.009 4.082

As more strings are created (with references kept to them), the list creation operations slow down only a bit in pqR, but they slow down enormously in R-3.5.1, as every garbage collection (even partial ones) has to scan an increasing number of character strings.