Supplementary material for:

dotCall64: An R package providing an efficient interface to compiled C, C++, and Fortran code supporting long vectors

Florian Gerber, Kaspar Mösinger, Reinhard Furrer

S1 Technical note on the underlying C implementations of long vectors in R

We refer to the source code of R version 3.3.1 in several places and show relevant parts thereof below. Information on the current and future directions of long vectors and 64-bit types in R can be found in "R Internals" (R Core Team, 2018a, Section 12).

In R, vectors are made out of a header of type *VECSEXP* that is followed by the actual data (Listing 1, line 272). The header contains a field *length* of type *R_len_t*, which is defined as signed *int32_t* (a 32-bit integer). Thus, that *length* field cannot capture the length of a long vector. Instead, it is set to *-1* whenever the length of the vector is larger than $2^{31} - 1$, and an additional header of type *R_long_vec_hdr_t* is prefixed. The prefixed header has a field *length* of type *R_xlen_t*, which is defined as *ptrdiff_t* type (Listing 1, line 75) being "the signed *integer* type of the result of subtracting two pointers. This will probably be one of the standard signed *integer* types (short int, int or long int), but might be a nonstandard type that exists only for this purpose" (GNU C Library, 2016, Appendix A.4).

This implementation has the advantage that the existing code does not need to be changed and still works with vectors having less than 2^{31} elements. Hence, the C code of R can be changed successively to support long vectors throughout several R versions, as opposed to changing the entire C code in one step. To make C code compatible with long vectors, adaptations are needed. For example, the widely used C function *R_len_t length(SEXP s)* (Listing 2, line 124) returns the length of a *SEXP* (S expression) as a *R_len_t*. Thus, all instances of that function have to be replaced with calls to the 64-bit counterpart (i. e., the function *R_xlen_t xlength(SEXP s)* given in line 159 of Listing 2).

Listing 1: *R-3.3.1/src/include/Rinternals.h*

```
26 #ifndef R_INTERNALS_H_
27 #define R_INTERNALS_H_
28
29 // Support for NO_C_HEADERS added in R 3.3.0
30 #ifdef __cplusplus
31 # ifndef NO_C_HEADERS
32 #  include <cstdio>
```

```
#  ifdef __SUNPRO_CC
using std::FILE;
#  endif
#  include <climits>
#  include <cstddef>
# endif
extern "C" {
#else
# ifndef NO_C_HEADERS
#  include <stdio.h>
#  include <limits.h> /* for INT_MAX */
#  include <stddef.h> /* for ptrdiff_t */
# endif
#endif

#include <R_ext/Arith.h>
#include <R_ext/Boolean.h>
#include <R_ext/Complex.h>
#include <R_ext/Error.h>  // includes NORET macro
#include <R_ext/Memory.h>
#include <R_ext/Utils.h>
#include <R_ext/Print.h>

#include <R_ext/libextern.h>

typedef unsigned char Rbyte;

/* type for length of (standard, not long) vectors etc */
typedef int R_len_t;
#define R_LEN_T_MAX INT_MAX

/* both config.h and Rconfig.h set SIZEOF_SIZE_T, but Rconfig.h is
   skipped if config.h has already been included. */
#ifndef R_CONFIG_H
# include <Rconfig.h>
#endif

#if ( SIZEOF_SIZE_T > 4 )
# define LONG_VECTOR_SUPPORT
#endif

#ifdef LONG_VECTOR_SUPPORT
    typedef ptrdiff_t R_xlen_t;
    typedef struct { R_xlen_t lv_length, lv_truelength; } R_long_vec_hdr_t;
# define R_XLEN_T_MAX 4503599627370496
# define R_SHORT_LEN_MAX 2147483647
# define R_LONG_VEC_TOKEN -1
#else
    typedef int R_xlen_t;
# define R_XLEN_T_MAX R_LEN_T_MAX
#endif

#ifndef TESTING_WRITE_BARRIER
# define INLINE_PROTECT
#endif

/* Fundamental Data Types:  These are largely Lisp
 * influenced structures, with the exception of LGLSXP,
 * INTSXP, REALSXP, CPLXSXP and STRSXP which are the
 * element types for S-like data objects.
 *
 *   --> TypeTable[] in ../main/util.c for  typeof()
 */

/*  These exact numeric values are seldom used, but they are, e.g., in
 *  ../main/subassign.c, and they are serialized.
*/
#ifndef enum_SEXPTYPE
/* NOT YET using enum:
 *  1) The SEXPREC struct below has 'SEXPTYPE type : 5'
 * (making FUNSXP and CLOSXP equivalent in there),
```

```
 * giving (-Wall only ?) warnings all over the place
 * 2) Many switch(type) { case ... } statements need a final `default:'
 * added in order to avoid warnings like [e.g. l.170 of ../main/util.c]
 *   "enumeration value `FUNSXP' not handled in switch"
 */
typedef unsigned int SEXPTYPE;

#define NILSXP       0    /* nil = NULL */
#define SYMSXP       1    /* symbols */
#define LISTSXP      2    /* lists of dotted pairs */
#define CLOSXP       3    /* closures */
#define ENVSXP       4    /* environments */
#define PROMSXP      5    /* promises: [un]evaluated closure arguments */
#define LANGSXP      6    /* language constructs (special lists) */
#define SPECIALSXP   7    /* special forms */
#define BUILTINSXP   8    /* builtin non-special forms */
#define CHARSXP      9    /* "scalar" string type (internal only)*/
#define LGLSXP      10    /* logical vectors */
/* 11 and 12 were factors and ordered factors in the 1990s */
#define INTSXP      13    /* integer vectors */
#define REALSXP     14    /* real variables */
#define CPLXSXP     15    /* complex variables */
#define STRSXP      16    /* string vectors */
#define DOTSXP      17    /* dot-dot-dot object */
#define ANYSXP      18    /* make "any" args work.
         Used in specifying types for symbol
         registration to mean anything is okay  */
#define VECSXP      19    /* generic vectors */
#define EXPRSXP     20    /* expressions vectors */
#define BCODESXP    21    /* byte code */
#define EXTPTRSXP   22    /* external pointer */
#define WEAKREFSXP  23    /* weak reference */
#define RAWSXP      24    /* raw bytes */
#define S4SXP       25    /* S4, non-vector */

/* used for detecting PROTECT issues in memory.c */
#define NEWSXP      30    /* fresh node created in new page */
#define FREESXP     31    /* node released by GC */

#define FUNSXP      99    /* Closure or Builtin or Special */


#else /* NOT YET */
/*------ enum_SEXPTYPE ----- */
typedef enum {
    NILSXP = 0, /* nil = NULL */
    SYMSXP = 1, /* symbols */
    LISTSXP = 2, /* lists of dotted pairs */
    CLOSXP = 3, /* closures */
    ENVSXP = 4, /* environments */
    PROMSXP = 5, /* promises: [un]evaluated closure arguments */
    LANGSXP = 6, /* language constructs (special lists) */
    SPECIALSXP = 7, /* special forms */
    BUILTINSXP = 8, /* builtin non-special forms */
    CHARSXP = 9, /* "scalar" string type (internal only)*/
    LGLSXP = 10, /* logical vectors */
    INTSXP = 13, /* integer vectors */
    REALSXP = 14, /* real variables */
    CPLXSXP = 15, /* complex variables */
    STRSXP = 16, /* string vectors */
    DOTSXP = 17, /* dot-dot-dot object */
    ANYSXP = 18, /* make "any" args work */
    VECSXP = 19, /* generic vectors */
    EXPRSXP = 20, /* expressions vectors */
    BCODESXP = 21, /* byte code */
    EXTPTRSXP = 22, /* external pointer */
    WEAKREFSXP = 23, /* weak reference */
    RAWSXP = 24, /* raw bytes */
    S4SXP = 25, /* S4 non-vector */

    NEWSXP      = 30,   /* fresh node creaed in new page */
```

```c
    FREESXP     = 31,    /* node released by GC */

    FUNSXP = 99 /* Closure or Builtin */
} SEXPTYPE;
#endif

/* These are also used with the write barrier on, in attrib.c and util.c */
#define TYPE_BITS 5
#define MAX_NUM_SEXPTYPE (1<<TYPE_BITS)

// ====================================== USE_RINTERNALS section
#ifdef USE_RINTERNALS
/* This is intended for use only within R itself.
 * It defines internal structures that are otherwise only accessible
 * via SEXP, and macros to replace many (but not all) of accessor functions
 * (which are always defined).
 */

/* Flags */


struct sxpinfo_struct {
    SEXPTYPE type      :  TYPE_BITS;/* ==> (FUNSXP == 99) %% 2^5 == 3 == CLOSXP
        * -> warning: `type' is narrower than values
        *              of its type
        * when SEXPTYPE was an enum */
    unsigned int obj   :  1;
    unsigned int named :  2;
    unsigned int gp    : 16;
    unsigned int mark  :  1;
    unsigned int debug :  1;
    unsigned int trace :  1;  /* functions and memory tracing */
    unsigned int spare :  1;  /* currently unused */
    unsigned int gcgen :  1;  /* old generation number */
    unsigned int gccls :  3;  /* node class */
}; /*       Tot: 32 */

struct vecsxp_struct {
    R_len_t length;
    R_len_t truelength;
};

struct primsxp_struct {
    int offset;
};

struct symsxp_struct {
    struct SEXPREC *pname;
    struct SEXPREC *value;
    struct SEXPREC *internal;
};

struct listsxp_struct {
    struct SEXPREC *carval;
    struct SEXPREC *cdrval;
    struct SEXPREC *tagval;
};

struct envsxp_struct {
    struct SEXPREC *frame;
    struct SEXPREC *enclos;
    struct SEXPREC *hashtab;
};

struct closxp_struct {
    struct SEXPREC *formals;
    struct SEXPREC *body;
    struct SEXPREC *env;
};

struct promsxp_struct {
```

```
    struct SEXPREC *value;
    struct SEXPREC *expr;
    struct SEXPREC *env;
};

/* Every node must start with a set of sxpinfo flags and an attribute
   field. Under the generational collector these are followed by the
   fields used to maintain the collector's linked list structures. */

/* Define SWITH_TO_REFCNT to use reference counting instead of the
   'NAMED' mechanism. This uses the R-devel binary layout. The two
   'named' field bits are used for the REFCNT, so REFCNTMAX is 3. */
//#define SWITCH_TO_REFCNT

#if defined(SWITCH_TO_REFCNT) && ! defined(COMPUTE_REFCNT_VALUES)
# define COMPUTE_REFCNT_VALUES
#endif
#define REFCNTMAX (4 - 1)

#define SEXPREC_HEADER \
    struct sxpinfo_struct sxpinfo; \
    struct SEXPREC *attrib; \
    struct SEXPREC *gengc_next_node, *gengc_prev_node

/* The standard node structure consists of a header followed by the
   node data. */
typedef struct SEXPREC {
    SEXPREC_HEADER;
    union {
	struct primsxp_struct primsxp;
	struct symsxp_struct symsxp;
	struct listsxp_struct listsxp;
	struct envsxp_struct envsxp;
	struct closxp_struct closxp;
	struct promsxp_struct promsxp;
    } u;
} SEXPREC, *SEXP;

/* The generational collector uses a reduced version of SEXPREC as a
   header in vector nodes.  The layout MUST be kept consistent with
   the SEXPREC definition.  The standard SEXPREC takes up 7 words on
   most hardware; this reduced version should take up only 6 words.
   In addition to slightly reducing memory use, this can lead to more
   favorable data alignment on 32-bit architectures like the Intel
   Pentium III where odd word alignment of doubles is allowed but much
   less efficient than even word alignment. */
typedef struct VECTOR_SEXPREC {
    SEXPREC_HEADER;
    struct vecsxp_struct vecsxp;
} VECTOR_SEXPREC, *VECSEXP;

typedef union { VECTOR_SEXPREC s; double align; } SEXPREC_ALIGN;

/* General Cons Cell Attributes */
#define ATTRIB(x)	((x)->attrib)
#define OBJECT(x)	((x)->sxpinfo.obj)
#define MARK(x)		((x)->sxpinfo.mark)
#define TYPEOF(x)	((x)->sxpinfo.type)
#define NAMED(x)	((x)->sxpinfo.named)
#define RTRACE(x)	((x)->sxpinfo.trace)
#define LEVELS(x)	((x)->sxpinfo.gp)
#define SET_OBJECT(x,v)	(((x)->sxpinfo.obj)=(v))
#define SET_TYPEOF(x,v)	(((x)->sxpinfo.type)=(v))
#define SET_NAMED(x,v)	(((x)->sxpinfo.named)=(v))
#define SET_RTRACE(x,v)	(((x)->sxpinfo.trace)=(v))
#define SETLEVELS(x,v)	(((x)->sxpinfo.gp)=((unsigned short)v))

#if defined(COMPUTE_REFCNT_VALUES)
# define REFCNT(x) ((x)->sxpinfo.named)
# define TRACKREFS(x) (TYPEOF(x) == CLOSXP ? TRUE : ! (x)->sxpinfo.spare)
#else
```

```c
# define REFCNT(x) 0
# define TRACKREFS(x) FALSE
#endif

#ifdef SWITCH_TO_REFCNT
# undef NAMED
# undef SET_NAMED
# define NAMED(x) REFCNT(x)
# define SET_NAMED(x, v) do {} while (0)
#endif

/* S4 object bit, set by R_do_new_object for all new() calls */
#define S4_OBJECT_MASK ((unsigned short)(1<<4))
#define IS_S4_OBJECT(x) ((x)->sxpinfo.gp & S4_OBJECT_MASK)
#define SET_S4_OBJECT(x) (((x)->sxpinfo.gp) |= S4_OBJECT_MASK)
#define UNSET_S4_OBJECT(x) (((x)->sxpinfo.gp) &= ~S4_OBJECT_MASK)

/* Vector Access Macros */
#ifdef LONG_VECTOR_SUPPORT
    R_len_t NORET R_BadLongVector(SEXP, const char *, int);
# define IS_LONG_VEC(x) (SHORT_VEC_LENGTH(x) == R_LONG_VEC_TOKEN)
# define SHORT_VEC_LENGTH(x) (((VECSEXP) (x))->vecsxp.length)
# define SHORT_VEC_TRUELENGTH(x) (((VECSEXP) (x))->vecsxp.truelength)
# define LONG_VEC_LENGTH(x) ((R_long_vec_hdr_t *) (x))[-1].lv_length
# define LONG_VEC_TRUELENGTH(x) ((R_long_vec_hdr_t *) (x))[-1].lv_truelength
# define XLENGTH(x) (IS_LONG_VEC(x) ? LONG_VEC_LENGTH(x) : SHORT_VEC_LENGTH(x))
# define XTRUELENGTH(x) (IS_LONG_VEC(x) ? LONG_VEC_TRUELENGTH(x) : SHORT_VEC_TRUELENGTH(x))
# define LENGTH(x) (IS_LONG_VEC(x) ? R_BadLongVector(x, __FILE__, __LINE__) : SHORT_VEC_LENGTH(x))
# define TRUELENGTH(x) (IS_LONG_VEC(x) ? R_BadLongVector(x, __FILE__, __LINE__) : SHORT_VEC_TRUELENGTH(x))
# define SET_SHORT_VEC_LENGTH(x,v) (SHORT_VEC_LENGTH(x) = (v))
# define SET_SHORT_VEC_TRUELENGTH(x,v) (SHORT_VEC_TRUELENGTH(x) = (v))
# define SET_LONG_VEC_LENGTH(x,v) (LONG_VEC_LENGTH(x) = (v))
# define SET_LONG_VEC_TRUELENGTH(x,v) (LONG_VEC_TRUELENGTH(x) = (v))
# define SETLENGTH(x,v) do { \
      SEXP sl__x__ = (x); \
      R_xlen_t sl__v__ = (v); \
      if (IS_LONG_VEC(sl__x__)) \
   SET_LONG_VEC_LENGTH(sl__x__,  sl__v__); \
      else SET_SHORT_VEC_LENGTH(sl__x__, (R_len_t) sl__v__); \
  } while (0)
# define SET_TRUELENGTH(x,v) do { \
      SEXP sl__x__ = (x); \
      R_xlen_t sl__v__ = (v); \
      if (IS_LONG_VEC(sl__x__)) \
   SET_LONG_VEC_TRUELENGTH(sl__x__, sl__v__); \
      else SET_SHORT_VEC_TRUELENGTH(sl__x__, (R_len_t) sl__v__); \
  } while (0)
# define IS_SCALAR(x, type) (TYPEOF(x) == (type) && SHORT_VEC_LENGTH(x) == 1)
#else
# define SHORT_VEC_LENGTH(x) (((VECSEXP) (x))->vecsxp.length)
# define LENGTH(x) (((VECSEXP) (x))->vecsxp.length)
# define TRUELENGTH(x) (((VECSEXP) (x))->vecsxp.truelength)
# define XLENGTH(x) LENGTH(x)
# define XTRUELENGTH(x) TRUELENGTH(x)
# define SETLENGTH(x,v)  ((((VECSEXP) (x))->vecsxp.length)=(v))
# define SET_TRUELENGTH(x,v) ((((VECSEXP) (x))->vecsxp.truelength)=(v))
# define SET_SHORT_VEC_LENGTH SETLENGTH
# define SET_SHORT_VEC_TRUELENGTH SET_TRUELENGTH
# define IS_LONG_VEC(x) 0
# define IS_SCALAR(x, type) (TYPEOF(x) == (type) && LENGTH(x) == 1)
#endif
```

Listing 2: *R-3.3.1/src/include/Rinlinedfuns.h*

```c
INLINE_FUN R_len_t length(SEXP s)
{
    switch (TYPEOF(s)) {
    case NILSXP:
 return 0;
    case LGLSXP:
```

```
    case INTSXP:
    case REALSXP:
    case CPLXSXP:
    case STRSXP:
    case CHARSXP:
    case VECSXP:
    case EXPRSXP:
    case RAWSXP:
 return LENGTH(s);
    case LISTSXP:
    case LANGSXP:
    case DOTSXP:
    {
 int i = 0;
 while (s != NULL && s != R_NilValue) {
     i++;
     s = CDR(s);
 }
 return i;
    }
    case ENVSXP:
 return Rf_envlength(s);
    default:
 return 1;
    }
}

R_xlen_t Rf_envxlength(SEXP rho);

INLINE_FUN R_xlen_t xlength(SEXP s)
{
    switch (TYPEOF(s)) {
    case NILSXP:
 return 0;
    case LGLSXP:
    case INTSXP:
    case REALSXP:
    case CPLXSXP:
    case STRSXP:
    case CHARSXP:
    case VECSXP:
    case EXPRSXP:
    case RAWSXP:
 return XLENGTH(s);
    case LISTSXP:
    case LANGSXP:
    case DOTSXP:
    {
 // it is implausible this would be >= 2^31 elements, but allow it
 R_xlen_t i = 0;
 while (s != NULL && s != R_NilValue) {
     i++;
     s = CDR(s);
 }
 return i;
    }
    case ENVSXP:
 return Rf_envxlength(s);
    default:
 return 1;
    }
}
```

S2 Performance

S2.1 Performance relevant arguments of *.C64()*

.C64() provides arguments to optimize calls to compiled code, one of which is the argument *INTENT*, which is set to "read and write" by default. Since many compiled functions/subroutines only read or write to certain arguments, it is safe to avoid copying in some cases. For example, the C function *get64_c()*, as defined in the manuscript, only reads the arguments *input* and *index* and only writes to the argument *output*. Thus, we can set the *INTENT* argument of *.C64()* to *c("r", "r", "w")* and pass the argument with intent "write" as objects of class *"vector_dc"* to reduce the copying of R vectors to a minimum. Another significant performance gain is obtained by setting the argument *NAOK* to *TRUE*. This avoids checking the R vectors passed through *"..."* for *NA*, *NaN*, and *Inf* values. Small-scale performance gains can be achieved by setting the *PACKAGE* argument, which reduces the time to find the compiled code, and by setting *VERBOSE = 0*, which avoids the execution of *getOptions("dotCall64.verbose")*. Additional speed improvements as described in "Writing R Extensions" (R Core Team, 2018b, Section 5.4.1) are partially applicable to *.C64()*. An optimized version of the call to the C function *get64_c()*, taking the discussed performance considerations into account, is given next.

```
R> .C64("get64_c", SIGNATURE = c("double", "int64", "double"),
        input = x_long, index = 2^31, output = numeric_dc(1),
        INTENT = c("r", "r", "w"), NAOK = TRUE, PACKAGE = "dotCall64", VERBOSE = 0)
```

S2.2 Timing measurements

In the following, we present detailed timing measurements and benchmark *.C64()* against *.C()*, where possible. We consider the following C function contained in the R package dotCall64.

```
void BENCHMARK(void *a) { }
```

This function takes one pointer *a* to a variable of an unspecified data type and does no operations with it. Thus, the elapsed time to call *BENCHMARK()* from R is dominated by the performance of the used interface. We measure the time to call this function with different *NAOK* and *INTENT* settings of *.C64()* and benchmark it against *.C()* using microbenchmark (Mersmann et al., 2018). To get an estimate of the measurement uncertainty, we repeated the measurements between 100 and 10′000 times and report the median elapsed time as well as the interquartile range (IQR) of the replicates. Naturally, timing measurements are platform dependent. We produced the presented results on Intel Xeon CPU E7-2850 2.00 GHz processors using a 64-bit Linux environment where R was installed with default installation flags. When not indicated differently, the measurements were produced using a single thread.

First, we consider the situation in which a pointer to an R vector of length one is passed to the compiled C function *BENCHMARK()*. The following truncated R code illustrates how the measurements were performed. The complete R scripts implementing all presented performance measurements are available in the *benchmark* directory in the source code of dotCall64.

```
R> library("microbenchmark")
R> int <- integer(1)
R> microbenchmark(
    .C("BENCHMARK", a = int, NAOK = FALSE, PACKAGE = "dotCall64"),
    .C64("BENCHMARK", SIGNATURE = "integer", a = int, INTENT = "rw",
         NAOK = FALSE, PACKAGE = "dotCall64", VERBOSE = 0),
    .C64("BENCHMARK", SIGNATURE = "integer", a = int, INTENT = "r",
         NAOK = FALSE, PACKAGE = "dotCall64", VERBOSE = 0),
...
```

Since the R vector *int* is very short, a large part of the elapsed time in this experiment is caused by the overhead of the interfaces. Table S1 presents the resulting timing measurements in microseconds. They indicate that *.C()* is more than two times faster compared to *.C64()*. However, this is not surprising, since *.C64()* is more flexible and therefore has a larger overhead. The arguments *NAOK* and *INTENT* have little influence on the elapsed times. The IQRs of around one microsecond indicate a relatively large variability of the elapsed time, which is typical for short timing measurements.

Table S1: Elapsed times in microseconds to pass double, integer, and 64-bit integer pointers to vectors of length one from R to C using *.C()* and *.C64()*. The used *INTENT* arguments of *.C64()* are indicated in brackets. Reported are median elapsed times of 10′000 replicates. The corresponding IQRs are indicated in parentheses.

	NAOK = FALSE			*NAOK = TRUE*		
	.C	.C64 [rw]	.C64 [r]	.C	.C64 [rw]	.C64 [r]
double	2.43 (0.46)	7.11 (0.37)	6.97 (0.40)	2.40 (0.45)	7.04 (0.35)	6.92 (0.37)
integer	2.39 (0.33)	7.54 (0.85)	7.43 (0.85)	2.39 (0.34)	7.52 (0.84)	7.39 (0.83)
64-bit integer		8.98 (1.14)	8.63 (1.19)		8.91 (1.17)	8.58 (1.17)

We repeat the same experiment with vectors of length 2^{28} (Table S2). Now, the elapsed times are dominated by services of the interfaces (i. e., checking for missing/infinite values, copying, and casting). They indicate that checking for missing/infinite values (*NAOK = FALSE*) increases the elapsed times across all considered cases. Moreover, *.C64()* with argument *INTENT = "rw"* and *.C()* show similar elapsed times. When the intent is set to "read" (*INTENT = "r"*), the elapsed times are reduced and dropped to microseconds seconds for some configurations. The castings of *SIGNATURE = "int64"* arguments seems to be the most time-consuming task. Note that the IQRs are now smaller relative to the measured timings, because the measured times are larger.

Table S2: Elapsed times in seconds to pass double, integer, and 64-bit integer pointers to vectors of length 2^{28} from R to C using *.C()* and *.C64()*. The used *INTENT* arguments of *.C64()* are indicated in brackets. Reported are median elapsed times of 100 replicates. The corresponding IQRs are indicated in parentheses.

	NAOK = FALSE			*NAOK = TRUE*		
	.C	.C64 [rw]	.C64 [r]	.C	.C64 [rw]	.C64 [r]
double	2.65 (0.05)	3.16 (0.06)	1.82 (0.02)	1.33 (0.06)	1.33 (0.05)	0.00 (0.00)
integer	1.09 (0.03)	1.09 (0.04)	0.43 (0.01)	0.66 (0.03)	0.66 (0.04)	0.00 (0.00)
64-bit integer		5.21 (0.20)	3.80 (0.06)		3.36 (0.06)	1.97 (0.06)

In another series of timing measurements, we consider the situation in which a pointer to a vector is passed to the compiled code to write into the vector. We measure the elapsed times of this task as shown in the following truncated R code.

```
R> microbenchmark(
     .C("BENCHMARK", a = integer(2^28), NAOK = TRUE, package = "dotCall64")
     .C64("BENCHMARK", SIGNATURE = "integer", a = integer(2^28), INTENT = "rw",
          NAOK = TRUE, package = "dotCall64", VERBOSE = 0)
     .C64("BENCHMARK", SIGNATURE = "integer", a = integer_dc(2^28), INTENT = "w",
          NAOK = TRUE, package = "dotCall64", VERBOSE = 0),
...
```

The results of this experiment are shown in Table S3. Note the usage of *integer_dc()*, which creates a list containing the length and class of the vector. This information is then used by *.C64()* to create the corresponding vector in C. Table S3 shows the timing measurements for the described setting. As expected using *.C64()* with *INTENT = "w"* reduces the elapsed times compared to *INTENT = "rw"* substantially. Furthermore, *.C()* and *.C64()* with *INTENT = "w"* have similar elapsed times. While *.C()* relies on the reference counting mechanism of R objects to avoid copying ("Writing R Extensions," R Core Team, 2018b), *.C64()* uses the *"vector_dc"* class. The latter has the advantage that one *double* to 64-bit integer casting can be avoided in the *SIGNATURE = "int64"* case.

Table S3: Elapsed times in seconds to pass double, integer, and 64-bit integer pointers to vectors of length 2^{28} initialized with zeros from R to C using *.C()* and *.C64()*. The used *INTENT* arguments of *.C64()* are indicated in brackets. Reported are median elapsed times of 100 replicates. The corresponding IQRs are indicated in parentheses.

	NAOK = TRUE		
	.C	.C64 [rw]	.C64 [w]
double	0.87 (0.01)	2.28 (0.13)	0.87 (0.01)
integer	0.44 (0.01)	1.16 (0.06)	0.44 (0.01)
64-bit integer		4.27 (0.03)	2.27 (0.02)

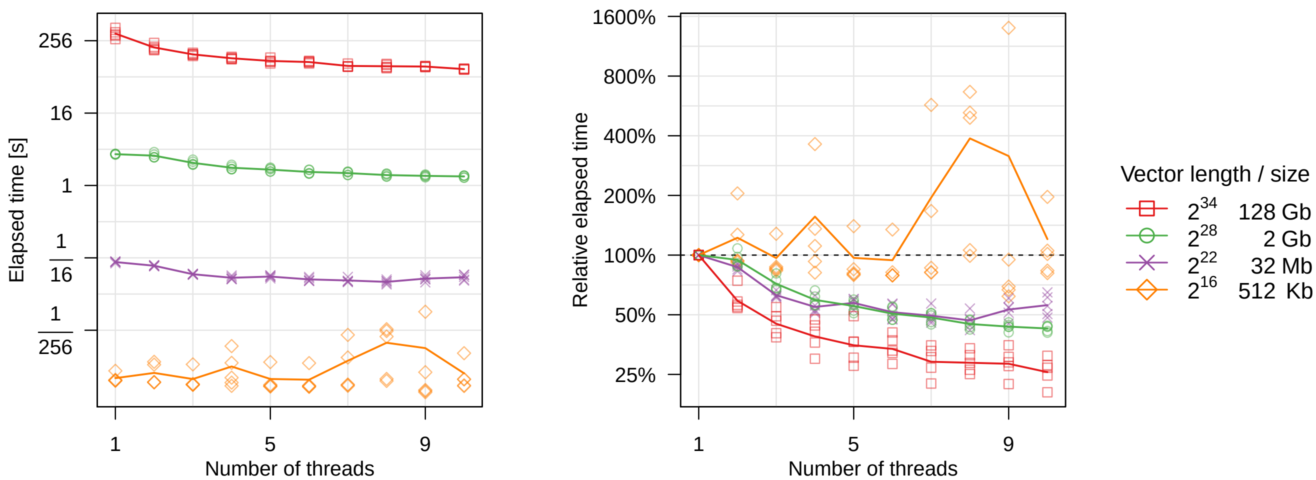

Figure S1: Timings measurements to illustrate the effect of using *.C64()* with enabled multithreading (openMP). Colors and symbols indicate the length/size of the evaluated vectors. Five replicates of each measured configuration are shown with symbols, and the mean values thereof are connected with a line. Left panel: The elapsed time in seconds (y-axis) is plotted against the number of used threads (x-axis). Right panel: The decrease/increase in elapsed time relative to using one thread (y-axis) is plotted against the number of threads (x-axis).

The function *.C64()* features an openMP implementation of the *double* to 64-bit integer and 64-bit integer to *double* castings of *SIGNATURE = "int64"* arguments. Hence, the computational workload of the castings can be distributed to several threads running in parallel. To quantify the performance gain related to using openMP, we control the number of used threads to be between 1 and 10 with the R package OpenMPController and measure the elapsed times of the following call.

```
R> .C64("BENCHMARK", SIGNATURE = "int64", a = a, INTENT = "rw", NAOK = TRUE,
        PACKAGE = "dotCall64", VERBOSE = 0)
```

We let a be *double* vectors of length 2^{16}, 2^{22}, 2^{28}, and 2^{34} and performed five replicated timing measurements for each configuration. The results are summarized in Figure S1. The reduction in computation time due to using multiple threads is greatest for the vectors of length 2^{34}, where using 10 threads reduced the elapsed times by about 70%. Conversely, for the vector of length 2^{16} no reduction was observed.

S3 Fortran example

The function *.C64()* can also be used to interface compiled Fortran code. To highlight some Fortran specific features, we translate the C function *get_c()* into the Fortran subroutine *get_f()*.

```
R> writeLines("
      subroutine get_f(input, index, output)
      double precision :: input(*), output(*)
      integer :: index
      output(1) = input(index)
      end
", con = "get_f.f")
```

Note that we only use lower case letters in the Fortran subroutine and variable names to avoid unnecessary symbol-name translations. We compile the subroutine with the command line command *R CMD SHLIB get_f.f* to obtain the dynamic shared object (*get_f.so* on our platform). In contrast to *.Fortran()*, *.C64()* allows passing pointers to long vectors.

```
R> dyn.load(paste0("get_f", .Platform$dynlib.ext))
R> .C64("get_f", SIGNATURE = c("double", "integer", "double"),
        input = x_long, index = 9, output = double(1))$output
[1] 9
```

Again, elements with positions beyond $2^{31} - 1$ cannot be accessed, since the argument *index* is of type *integer* and compiled as a 32-bit integer by default. To make *get_f()* compatible with 64-bit integers, we can either change the declaration of *index* to *integer (kind = 8) index* in *get_f.f* or leave the Fortran code unchanged and set the following compiler flag to compile *integers* as 64-bit integers.

```
MAKEFLAGS="PKG_FFLAGS=-fdefault-integer-8" R CMD SHLIB get_f.f
```

Note that both the *kind = 8* declaration and the *-fdefault-integer-8* flag are valid for the GFortran compiler (GNU Fortran compiler, 2014) and may not have the intended effect using other compilers. The resulting dynamic shared object from the command above (*get_f.so* on our platform) can be called from R as follows.

```
R> dyn.load(paste0("get_f", .Platform$dynlib.ext))
R> .C64("get_f", SIGNATURE = c("double", "int64", "double"),
        input = x_long, index = 2^31, output = double(1))$output
[1] -1
```

The same call optimized by the specification of the *INTENT* argument.

```
R> .C64("get_f", SIGNATURE = c("double", "int64", "double"),
        INTENT = c("r", "r", "w"), input = x_long, index = 2^31,
        output = vector_dc("numeric", 1))$output
[1] -1
```

S4 Extend R packages to support long vectors

Extending R packages to support long vectors allows developers to distribute compiled code featuring 64-bit integers with an R user interface. Given the popularity of R, this is a promising approach to make such software available to many users. With the function *.C64()*, the workload of extending an R package to support long vectors is reduced to the following tasks:

- replace the R function to call compiled code with *.C64()*,
- replace the 32-bit integer type declarations in the compiled code with a 64-bit integer declaration.

The latter task implies replacing all *int* type declarations in C, C++ code with *int64_t* type declarations and replacing all *integer* type declarations in Fortran code with *integer (kind = 8)*. In both cases, the replacements can be automatized (e. g., with the stream editor GNU sed, 2010). If the considered Fortran code does not explicitly declare the bits of the integers, an alternative approach is to set the compiler flag *-fdefault-integer-8* to compile integers as 64-bit integers using GFortran compilers. This is convenient because in that case the Fortran code does not need to be changed at all.

A more elaborate extension could feature two versions of the compiled code: one with 32-bit integers and the other one with 64-bit integers. Then, the R function can dispatch to either version according to the sizes of the involved vectors. This avoids *double* to 64-bit integer castings when only vectors with less than $2^{31}-1$ elements are involved. It is convenient to manage two versions of compiled code by putting them into two separate R packages. The first package includes the compiled code with 32-bit integers together with the R code and the documentation. This package can be used independently as long as no long vectors are involved. The second package can be seen as an add-on package and includes only the compiled code with integers declared as 64-bit integers. Thus, loading both packages enables long vector support. This separation into two packages has the advantage that the compiled functions featuring 32-bit integers and their 64-bit counterparts can have the same name. The desired function is then specified by setting the appropriate *PACKAGE* argument of *.C64()*. In a proof-of-concept, we extended the sparse matrix algebra R package spam to handle sparse matrices with more the $2^{31}-1$ non-zero elements (Gerber et al., 2017).

S5 R code from manuscript

```
## interface C code
cat("
void get_c(double *input, int *index, double *output) {
          output[0] = input[index[0] - 1];
}",
   file = "get_c.c")
system("R CMD SHLIB get_c.c")

dyn.load(paste0("get_c", .Platform$dynlib.ext))
x <- 1:10
.C("get_c", input = as.double(x), index = as.integer(9), output = double(1))$output

x_long <- double(2^31); x_long[9] <- 9; x_long[2^31] <- -1
.C("get_c",
   input = as.double(x_long), index = as.integer(9), output = double(1))$output

library("dotCall64")
.C64("get_c", SIGNATURE = c("double", "integer", "double"),
     input = x_long, index = 9, output = double(1))$output

cat("
#include <stdint.h>
void get64_c(double *input, int64_t *index, double *output) {
output[0] = input[index[0] - 1];
}
",
    file = "get64_c.c")
system("R CMD SHLIB get64_c.c")

dyn.load(paste0("get64_c", .Platform$dynlib.ext))
.C64("get64_c", SIGNATURE = c("double", "int64", "double"),
     input = x_long, index = 2^31, output = double(1))$output

.C64("get64_c", SIGNATURE = c("double", "int64", "double"),
     INTENT = c("r", "r", "w"), input = x_long, index = 2^31,
     output = vector_dc("numeric", 1))$output
```

```r
## interface Fortran code
cat("
      subroutine get_f(input, index, output)
      double precision :: input(*), output(*)
      integer :: index
      output(1) = input(index)
      end
",
    file = "get_f.f")
system("R CMD SHLIB get_f.f")

dyn.load(paste0("get_f", .Platform$dynlib.ext))

.C64("get_f", SIGNATURE = c("double", "integer", "double"),
     input = x_long, index = 9, output = double(1))$output

file.remove("get_f.so", "get_f.o")
system("MAKEFLAGS=\"PKG_FFLAGS=-fdefault-integer-8\" R CMD SHLIB get_f.f")

dyn.load(paste0("get_f", .Platform$dynlib.ext))
.C64("get_f", SIGNATURE = c("double", "int64", "double"),
     input = x_long, index = 2^31, output = double(1))$output

.C64("get_f", SIGNATURE = c("double", "int64", "double"),
     INTENT = c("r", "r", "w"), input = x_long, index = 2^31,
     output = vector_dc("numeric", 1))$output

## clean
file.remove("get_c.c", "get_c.o",
            paste0("get_c", .Platform$dynlib.ext),
            "get64_c.c", "get64_c.o",
            paste0("get64_c", .Platform$dynlib.ext),
            "get_f.f", "get_f.o",
            paste0("get_f", .Platform$dynlib.ext))
```

References

Gerber, F., Mösinger, K., and Furrer, R. (2017). Extending R packages to support 64-bit compiled code: An illustration with spam64 and GIMMS NDVI_{3g} data. *Comput. Geosci.*, 104:109–119, doi:10.1016/j.cageo.2016.11.015. S13

GNU C Library (2016). Reference manual. S1

GNU Fortran compiler (2014). Reference manual for GCC version 4.9.2. S12

GNU sed (2010). Reference manual. S13

Mersmann, O., Beleites, C., Hurling, R., Friedman, A., and Ulrich, J. M. (2018). *microbenchmark: Accurate timing functions*. R package version 1.4-3. S8

R Core Team (2018a). *R Internals*. R Foundation for Statistical Computing, Vienna, Austria. R version 3.5.0. S1

R Core Team (2018b). *Writing R Extensions*. R Foundation for Statistical Computing, Vienna, Austria. R version 3.5.0. S8, S10