RcppMsgPack: MessagePack Headers and Interface Functions for R

by Travers Ching and Dirk Eddelbuettel

Abstract MessagePack, or **MsgPack** for short, or when referring to the implementation, is an efficient binary serialization format for exchanging data between different programming languages. The **RcppMsgPack** package provides *R* with both the MessagePack *C++* header files, and the ability to access, create and alter MessagePack objects directly from *R*. The main driver functions of the R interface are two functions `msgpack_pack` and `msgpack_unpack`. The function `msgpack_pack` serializes *R* objects to a raw MessagePack message. The function `msgpack_unpack` de-serializes MessagePack messages back into *R* objects. Several helper functions are available to aid in processing and formatting data including `msgpack_simplify`, `msgpack_format` and `msgpack_map`.

Introduction

MessagePack (or **MsgPack** for short, or when referring to the actual implementation) is a binary serialization format made for exchanging data between different programming languages (Furuhashi, 2018). Unlike other related formats such as JSON, **MsgPack** is a binary format—which makes it more efficient in terms of (disk or memory) space, transfer speeds (which is increasingly important for large data sets across networks) and potentially also precision (as textual representation rarely goes to the length of binary precision). As shown on the project homepage at `https://msgpack.org`, several major projects including *Redis*, *Pinterest*, *Fluentd* and *Treasure Data* utilize **MsgPack** to transfer data or to represent internal data structures (Furuhashi, 2018). Other binary serialization formats similar to **MsgPack** include BSON (MongoDB, 2018) and ProtoBuf (Google, 2018) which have their own advantages and disadvantages, such as serialization speed, memory usage, compression and requirement of descriptive schemas (Hamida et al., 2015; Dawborn and Curran, 2014). *R* support for these formats is available via the packages **mongolite** (Ooms, 2014) and **RProtoBuf** (Eddelbuettel et al., 2016); Redis is also implemented in *R* through the **RcppRedis** package (Eddelbuettel, 2017). **RcppMsgPack** brings support for the **MsgPack** specification to *R*.

The **MsgPack** specification describes a number of common data type: Booleans, Integers, Floats, Strings, Binary data, Arrays, Maps, and user-defined extension types, and has been implemented in most major programming languages. **RcppMsgPack** (Ching et al., 2018) aims to provide an efficient, and easy to use implementation by relying on the official *C++* **MsgPack** code and the **Rcpp** package (Eddelbuettel, 2013; Eddelbuettel et al., 2018). The package provides users with the **MsgPack** header files which can be used to more directly integrate **MsgPack** into *R* projects through *C++* code. It also provides the ability to serialize and de-serialize data directly to and from *R* (*e.g.*, through pipes, file handlers, sockets or binary object files). These functionalities can be used to efficiently transfer data between various programming languages and between separate *R* instances.

In this manuscript, we describe the main interface functions used to serialize and deserialize **MsgPack** messages, and the conversion between *R* data types and **MsgPack** data types. We describe helper functions contained in **RcppMsgPack** and describe several use cases and examples of how **RcppMsgPack** can be used in practice, benchmarking several common approaches for transferring data between processes.

Interface functions

The functions `msgpack_pack` and `msgpack_unpack` allow serialization and de-serialization of *R* objects respectively (Figure 1). Here, `msgpack_pack` takes in any number of *R* objects and generates a single serialized message in the form of a raw vector. Conversely, `msgpack_unpack` takes a serialized message as input, and returns the *R* object(s) contained in the message. Moreover, `msgpack_format` is a helper function to properly format *R* objects for input, and `msgpack_simplify` is a helper function to simplify output from a **MsgPack** conversion. One of the main goals of **MsgPack** is the transfer of data across processes and/or hosts. Therefore, we also define two helper functions `msgpack_write` and `msgpack_read` which facilitate writing and reading of **MsgPack** objects to files, pipes or any connection object.

The data types in **MsgPack** do not directly map on to *R* data types, more so than other languages. For example, basic *R* "atomic" types such as integers and strings are inherently vectorized, which is not true in *C++*, Python or most other languages. *I.e.*, in *R* there is no distinction between a single integer and a vector of integers of length 1. *R* also has multiple non-value types, such as `NULL` or `NA`

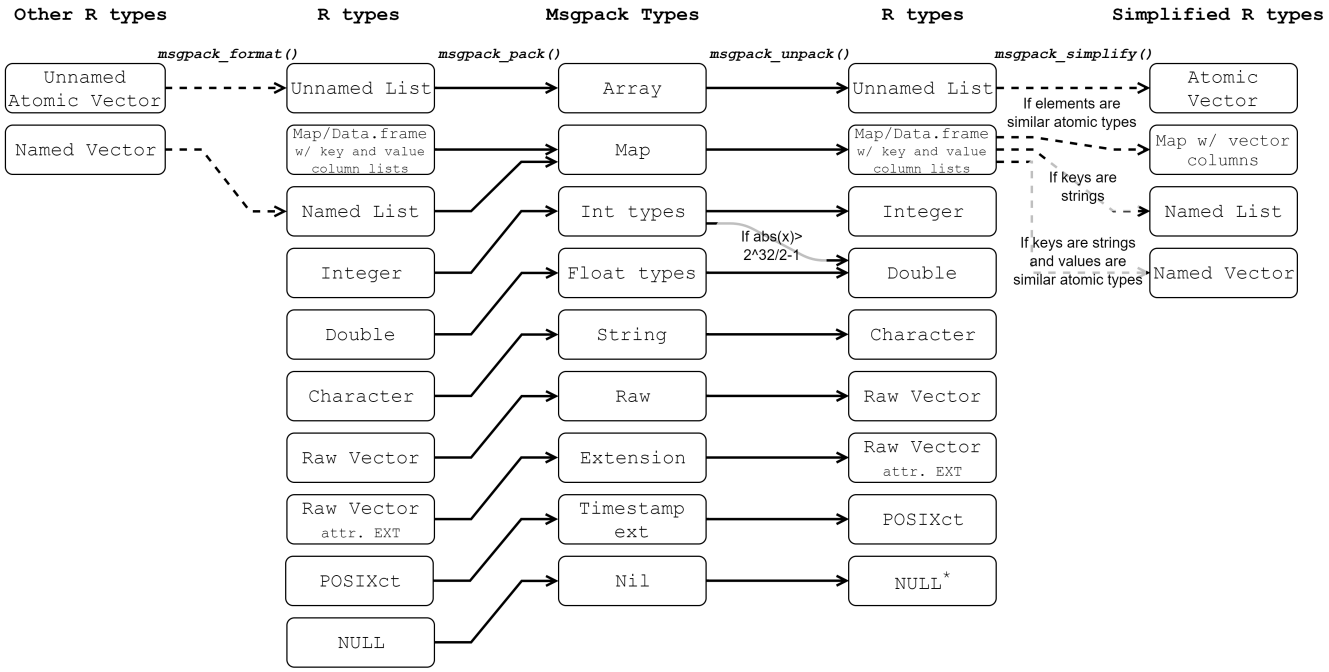

Figure 1: A flowchart of the conversion of *R* objects to **MsgPack** objects and vice versa.

for integer, string, numeric, etc. Because of these complexities, the conversion processes using these interface functions are described in detail below.

R integers are converted into **MsgPack** integers, which are automatically reduced in size, depending on the value of the integer. **MsgPack** integers are converted back into *R* integers. Because *R* does not natively support 64 bit integers, whereas **MsgPack** supports integers up to 64 unsigned bits in value, **MsgPack** integers exceeding signed 32 bits supported by *R* are coerced to *R* numeric values, with potential loss of precision. The integer `NA` value in *R* is represented by its bit value in *C++* (0x80000000), and requires no special treatment.

R `numeric` (*i.e.*, doubles) variables conform to IEEE 754 double-precision standards (IEEE Standards Committee, 2008), and also require no special treatment. The numeric NA value is a special case of NaN values, and is serialized by its bit representation.

R strings (*i.e.*, objects of class `character`) are converted to **MsgPack** strings. Because *C++* and **MsgPack** do not have missing values for strings, NA characters are converted into **MsgPack** Nil (similar to `NULL` in *R*).

R `logical` values are converted into **MsgPack** bool. Again, because NA logical values do not exist in *C++* or **MsgPack**, NA logical values are converted into **MsgPack** Nil.

R `raw` vectors are converted into **MsgPack** bin. Raw vectors with the "EXT" integer attribute are converted into **MsgPack** extension types. The EXT attribute should be a positive integer, as negative values are reserved for official extensions.

Currently, the **MsgPack** specifications includes one official extension type: timestamps. Timestamps are a **MsgPack** extension type with extension value -1 and can be converted to and from *R* `POSIXct` objects using `msgpack_timestamp_decode` and `msgpack_timestamp_encode` respectively. **MsgPack** timestamps can encode nanosecond precision. *R* `POSIXct` objects rely on `numeric`, and therefore conversion may have some loss of precision (unless a package such as **nanotime** (Eddelbuettel and Silvestri, 2017) is used, which is left as a future extension).

MsgPack specifications define two container objects: arrays and maps. **MsgPack** arrays are a sequential container object. The length of the array is defined in its message header. Arrays can contain any other **MsgPack** types, including other arrays or maps.

MsgPack arrays are naturally analogous to *R* `unnamed list` objects. However, because lists have a large memory footprint, *R* atomic vectors (with length of 0 or greater than or equal to 2) are also allowed as input for serialization to arrays.

MsgPack maps are an ordered sequence of key and value pairs, where each key and value can be any **MsgPack** object. There is no requirement for unique keys. Maps do not have an analogous data type in *R* . Therefore, maps are implemented by creating an object of class `map`, which is also a `data.frame` with `key` and `value` columns. As input to serialization, these columns can also be lists,

and can therefore contain any other *R* object, and not only a single type. The function `msgpack_map` is a simple helper function that takes two lists and returns a `map` which can be serialized into a **MsgPack** object with `msgpack_pack`.

In order to support as much generality as possible in serialization and deserialization, the use of lists to represent arrays and maps is necessary. However, it is often the case in *R* that one would want to deal with large vectors or matrices of a single type without the computational and memory overhead of lists. Two approaches are given to deal with this type of scenario. `msgpack_simplify` can be used after a call to `msgpack_unpack` to recursively simplify lists to vectors when only a single type is included within a list. (For lists of characters or logicals, this may also include NULLs.) Secondly, `msgpack_unpack` can be called with the `simplify=TRUE` parameter, which performs the same task as `msgpack_simplify` within *C++*, and is therefore much faster. The second approach can drastically improve speed and memory usage compared to the first approach.

Using MsgPack C++ headers through RcppMsgPack and Rcpp

Complex objects or data structures, such as trees, often do not fit into *R* data types because a tree data structure does not map nicely to an *R* vector, data.frame, matrix, etc. Storing such a complex object as a **MsgPack** message will be more performant in terms of serialization speed and memory usage.

The example below demonstrates how **MsgPack** headers can be integrated into a standard **Rcpp** workflow. In this example, a prefix tree is created for nucleotide sequences, and is serialized through **MsgPack** to create a persistant tree object in the form of a raw vector in *R*. The stored tree can be saved to disk, unpacked within *R* directly using `msgpack_unpack` or it can be reconstructed into the prefix tree within *C++* using the **MsgPack** *C++* interface. The code below defines a structure for storing the Prefix tree data and a function for constructing the tree using sequence data input from *R* and saving it as a **MsgPack** object:

```
struct Node {
  std::shared_ptr<Node> parent;
  std::set<int> sequence_idx;
  std::map< std::string, std::shared_ptr<Node> > children;
};

struct std::shared_ptr<Node>
NewNode(std::shared_ptr<Node> parent,
        std::string name,
        std::set<int> sequence_idx) {
  std::shared_ptr<Node> node = std::shared_ptr<Node>(new Node);
  node->sequence_idx = sequence_idx;
  if (parent) {
    parent->children.insert(std::pair<std::string,
                                      std::shared_ptr<Node> >(name, node));
  }
  return node;
}

void packTree(std::shared_ptr<Node> node,
              msgpack::packer<std::stringstream>& pkr) {
  pkr.pack_array(2);
  std::set<int> sequence_idx = node->sequence_idx;
  std::vector<int> vs(sequence_idx.begin(), sequence_idx.end());
  pkr.pack(vs);
  std::map< std::string, std::shared_ptr<Node> > children = node->children;
  pkr.pack_map(children.size());
  for (auto const& x : children) {
    pkr.pack(x.first);
    packTree(x.second, pkr);
  }
}

Rcpp::RawVector create_prefix_tree(std::vector<std::string> clone_sequences) {
  std::shared_ptr<Node> root;
  root = NewNode(std::shared_ptr<Node>(nullptr), "^", {});
  for(int i=0; i<clone_sequences.size(); i++) {
```

```
    std::shared_ptr<Node> current_node = root;
    for(int j=0; j < clone_sequences[i].size(); j++) {
      std::string nuc = clone_sequences[i].substr(j,1);
      if(current_node->children.count(nuc) == 1) {
        current_node = current_node->children[nuc];
        if(j == clone_sequences[i].size() - 1) {
          current_node->sequence_idx.insert(i);
        }
      } else {
        if(j == clone_sequences[i].size() - 1) {
          current_node = NewNode(current_node, nuc, {i});
        } else {
          current_node = NewNode(current_node, nuc, {});
        }
      }
    }
  }
  std::stringstream buffer;
  msgpack::packer<std::stringstream> pk(&buffer);
  packTree(root, pk);
  std::string bufstr = buffer.str();
  Rcpp::RawVector rawbuffer(bufstr.begin(), bufstr.end());
  return rawbuffer;
}
```

The `create_prefix_tree` function is an *C++* function that returns a raw vector, which is a serialization of the prefix tree. From *R*, the prefix tree can be initialized and serialized through calling the **Rcpp** function.

```
tree <- create_prefix_tree(c("AGCT", "AGCC", "AGC", "ATG", "GACC", "GTCT"))
```

Because the resulting message is a standard **MsgPack** object, the tree can be unpacked directly in *R* using `msgpack_unpack`. Additionally, the following code reconstructs the prefix tree from the **MsgPack** message from within *C++* using the C++ interface:

```
void print_node(msgpack::object & node_obj, std::string name) {
    std::vector<msgpack::object> node_obj_array;
    node_obj.convert(node_obj_array);
    msgpack::object_kv* p = node_obj_array[1].via.map.ptr;
    msgpack::object_kv* pend = node_obj_array[1].via.map.ptr +
                               node_obj_array[1].via.map.size;
    std::cout << name;
    if(p != pend) {
        std::cout << "(";
        for (; p < pend; ++p) {
            std::string name_child = p->key.as<std::string>();
            msgpack::object node_child_obj = p->val.as<msgpack::object>();
            print_node(node_child_obj, name_child);
            if(p < (pend-1)) std::cout << ",";
        }
        std::cout << ")";
    }
}
void print_newick_tree(std::vector<unsigned char> packed_tree) {
    std::string message(packed_tree.begin(), packed_tree.end());
    msgpack::object_handle oh;
    msgpack::unpack(oh, message.data(), message.size());
    msgpack::object obj = oh.get();
    print_node(obj, "Root");
}
```

Called from within *R*, the tree is printed to the console:

```
tree_nested_list <- msgpack_unpack(tree, simplify=F)
print_newick_tree(tree)

[1] Root(A(G(C(C,G,T),G)),G(A(C(C)),T(C(A,T))))
```

Writing MsgPack objects to disk and reading data from the internet

Following is an example of how one can create a **MsgPack** serialized message as a binary file, and read it from the internet into *R*. The first step would be to read in the data to *R* as normal, and serialize the data using the `msgpack_pack` function.

```
dat <- read.csv(paste0("https://raw.githubusercontent.com/vincentarelbundock/",
                       "Rdatasets/master/csv/mosaicData/Birthdays.csv"))
mp <- with(dat, msgpack_pack(X, state, year, month, day, date, wday, births))
```

The `msgpack_pack` function returns a raw vector, which can be written to disk using the `writeBin` function or the helper function `msgpack_write`.

```
msgpack_write(mp, file="birthdays_msgpack.mp")
```

A **MsgPack** object can be read directly from the internet, e.g., using the `GET` function from the **httr** package (Wickham, 2017). Subsequently, the `msgpack_unpack` function can be used to unpack the object to its original values.

```
md <- GET("http://travers.im/birthdays_msgpack.mp")
mp <- md$content
mu <- msgpack_unpack(mp)
```

Transferring large datasets from Python to R and back

To evaluate the performance of **MsgPack** serialization, we benchmarked the transfer of the MNIST (LeCun et al., 2010) and CIFAR-10 (Krizhevsky, 2009) datasets to and from Python and *R*. We compared this approach to writing and reading in CSV format, and to writing and reading using **feather** (Wickham et al., 2016), a cross-platform library and specification for efficiently handling tabular data. (The **feather** package is no longer actively developed, and will be superseded by Apache Arrow (Apache Arrow Developers, 2018), a more general cross-language development platform for working with in-memory data in a standardized language-independent columnar memory format. However, Arrow is not yet available for R.) In Python, serialization of the data was performed using the **msgpack** package and written to disk in binary format:

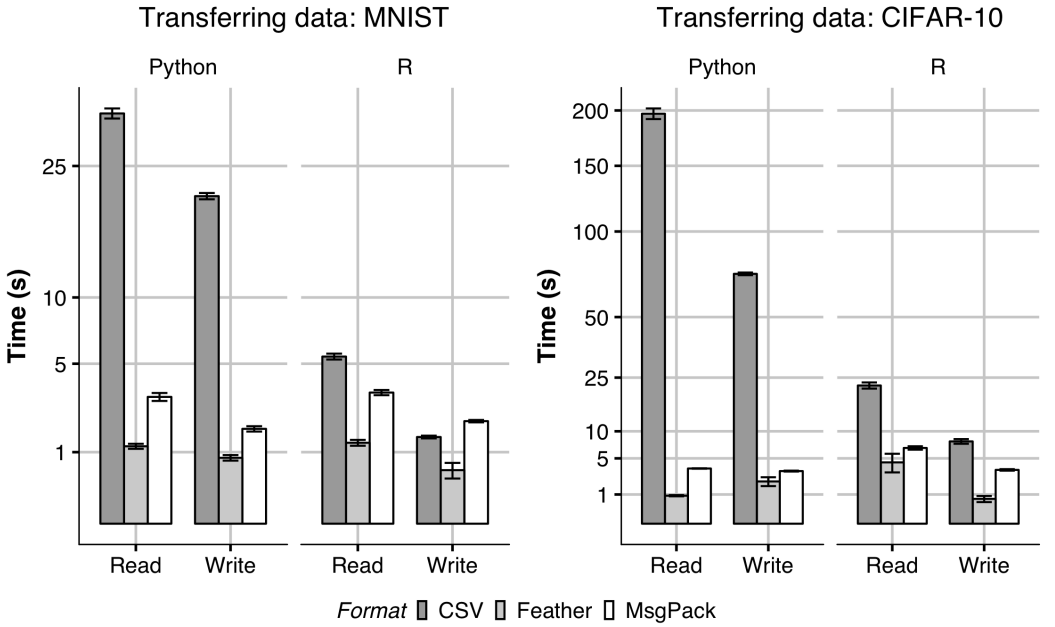

Figure 2: Transferring the MNIST dataset (left) and the CIFAR-10 dataset (right) to and from *R* and Python using either **MsgPack**, **feather** or CSV format.

```
xb = msgpack.packb(x)
with open("/tmp/dataset.mp", "wb") as file:
    file.write(xb)
```

Conversely, the `unpackb` function was used when benchmarking the transfer time from *R* to Python:

```
with open("/tmp/dataset.mp", "rb") as file:
    file.write(xb)
x = msgpack.unpackb(file.read())
```

In *R*, we used the `readBin` function followed by `msgpack_unpack` function to deserialize the dataset. Alternatively, one could use the helper function `msgpack_read`.

```
xb <- readBin(con = "/tmp/dataset.mp", "raw",
              n=file.info("/tmp/dataset.mp")$size)
x <- msgpack_unpack(xb, simplify=T)
```

Subsequently, the data can be re-serialized and written to disk using the `msgpack_pack` function followed by `writeBin`:

```
xb <- msgpack_pack(x)
writeBin(xb,"/tmp/dataset.mp", useBytes=T)
```

For CSV format, we used the `fread` and `fwrite` functions in the **data.table** package (Dowle et al., 2018) to write and read CSV tables in *R*. In Python, we used the package **numpy** `savetxt` and `loadtxt` functions (Walt et al., 2011). For the **feather** format, we used the `write_feather` and `read_feather` functions within the **feather** package in *R* and Python. All approaches were benchmarked after clearing the system cache and replicated 5 times (Figure 2).

Serialization and de-serialization of objects from **MsgPack** objects was similar between *R* and Python, with a slight speed advantage going to Python. Reading **MsgPack** objects was generally considerably faster than reading CSV format in both *R* and Python. Comparing **MsgPack** to **feather**, **feather** was generally faster. As **feather** is designed for columnar binary data, it does not use a header for every element and therefore has an advantage over the (more general) **MsgPack** in this context.

One surprising result is in writing of CSVs in R. Using **data.table**, writing the MNIST dataset was faster than **MsgPack**, although not quite as fast as **feather**. The MNIST dataset is formatted as a single floating point value in per pixel by **TensorFlow** (Abadi et al., 2016). When writing floating point data as CSV, both **numpy** and the **data.table** package truncate floating point numbers by default, which causes loss of precision.

MsgPack serialization memory usage

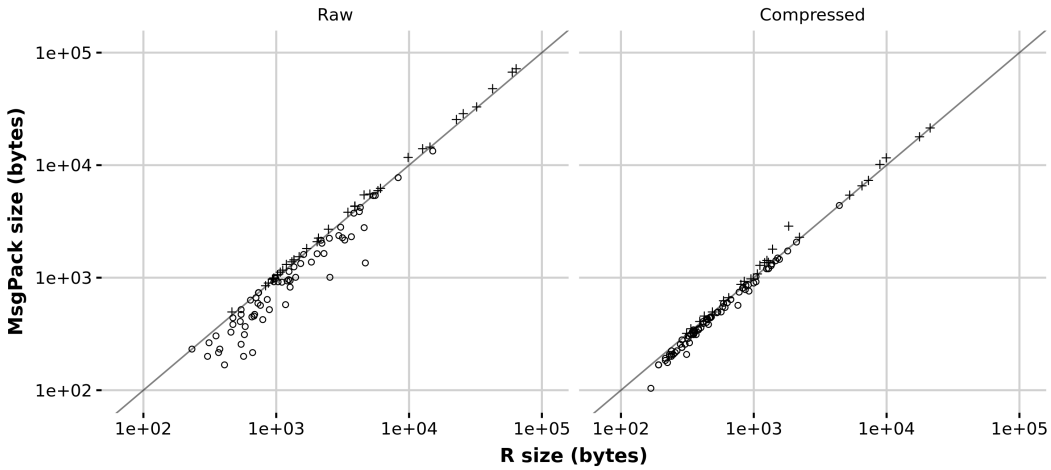

Figure 3: Serialization memory usage: **MsgPack** vs. *R*. Left: raw serialization size for datasets in **datasets**. Right: Gzip-compressed serialization size.

To compare the memory usage of **MsgPack** to native *R* serialization, we serialized the datasets found in the `datasets` *R* package (Figure 3). For *R* serialization, this was done by calling the `serialize` function for native *R* serialization or calling the `msgpack_pack` function for **MsgPack** serialization. For compression, the `memCompress` function was called on the results. Some datasets contained formulas,

or other *R*-specific attributes that can't be stored in **MsgPack**; these attributes were removed prior to serialization and compression.

Memory usage was generally very similar between **MsgPack** and native *R*. Although there were some differences in raw serialization, the difference seems to be less apparent after compression. For some uncompressed serializations, **MsgPack** significantly outperformed *R* in memory usage. This is attributable to efficient integer storage: **MsgPack** stores variable size integers depending on the integer magnitude, and can be as low as 8 bits. Native *R* serialization uses the modified External Data Representation (XDR) standard (Srinivasan, 1995), which uses a fixed 32 bits for integers. Furthermore, attributes of each dataset are stored as lists internally, which can increase the relative overhead for small datasets, as *R* lists are not memory-efficient objects.

On the other hand, every **MsgPack** element has a short header of several bits, which increases its memory overhead, particularly for large vectors. This overhead is apparent for larger datasets, where *R* typically is more memory efficient.

Serialization of large lists

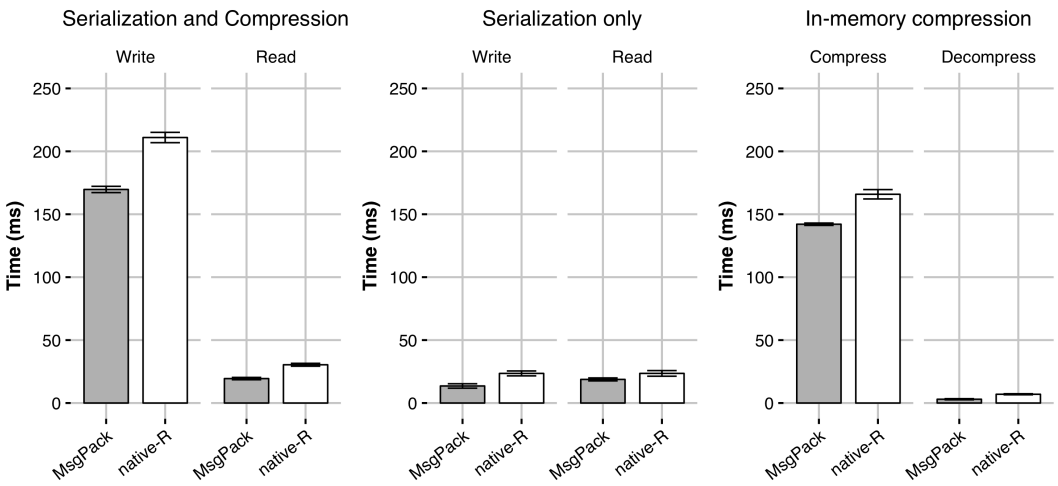

Figure 4: Serialization and compression time of a large list of DNA sequences.

As mentioned, one situation where **MsgPack** serialization is much more efficient than *R* serialization is how it handles list objects. Because R lists have a large memory overhead, both compression times and writing/reading to disk are faster using the **MsgPack** format. To show this, we generated a contrived list of DNA sequences as follows:

```
x <- replicate(10000, {
    replicate(sample(5,1), {
        paste(sample(c("G","C","A","T"), size=sample(50,1), replace=T), collapse="")
    })
})
```

If the data structure of the object is known, using the C++ headers directly can speed up serialization and de-serialization by avoiding the logic involved in serializing generic data structures, although the speed-up is not large. Compression and writing to disk can also be performed directly within C++, for example, by using the zlib C++ library (Gailly and Adler, 2018) to perform standard DEFLATE compression. The example below illustrates **MsgPack** serialization and subsequent compression using the **zlib** library:

```
void list_pack_gzip(List x, std::string file) {
  std::stringstream buffer;
  msgpack::packer<std::stringstream> pk(&buffer);
  pk.pack_array(x.size());
  for(int i=0; i<x.size(); i++) {
    CharacterVector xi = x[i];
    if(xi.size() == 1) {
      pk.pack(Rcpp::as<std::string>(xi[0]));
    } else {
```

```
      pk.pack_array(xi.size());
      for(int j=0; j<xi.size(); j++) {
        pk.pack(Rcpp::as<std::string>(xi[j]));
      }
    }
  }
  std::string bufstr = buffer.str();
  gzFile fi = gzopen(file.c_str(),"wb");
  gzwrite(fi, bufstr.c_str(), bufstr.size());
  gzclose(fi);
}
```

Conversely, uncompression and deserialization would need to be performed to recover the original R object. The following C++ function illustrates how this could be done:

```
Rcpp::List list_unpack_gzip(std::string file) {
  gzFile fi = gzopen(file.c_str(),"rb");
  char buf[8192];
  std::vector<char> dat;
  uint b = gzread(fi, buf, 8192);
  while (b) {
    dat.insert(dat.end(), buf, buf + b);
    b = gzread(fi, buf, 8192);
  }
  gzclose(fi);
  std::string message = std::string(dat.data(), dat.size());
  msgpack::object_handle oh;
  msgpack::unpack(oh, message.data(), message.size());
  msgpack::object obj = oh.get();
  std::vector<msgpack::object> obj_vector;
  obj.convert(obj_vector);
  Rcpp::List L = Rcpp::List(obj_vector.size());
  std::vector< std::string > temp_str_vec;
  std::string temp_str;
  for (int i=0; i< obj_vector.size(); i++) {
    if (obj_vector[i].type == msgpack::type::ARRAY) {
      obj_vector[i].convert(temp_str_vec);
      L[i] = Rcpp::wrap(temp_str_vec);
    } else {
      obj_vector[i].convert(temp_str);
      L[i] = Rcpp::wrap(temp_str);
    }
  }
  return(L);
}
```

To illustrate the potential benefit of this approach, we compared **MsgPack** serialization and **zlib** compression to standard R serialization using the `saveRDS` function (Figure 4). Using **MsgPack**, serialization is slightly faster for the example above. The timing of these two approaches can be broken down by process: both serialization and compression to disk (and vice versa) are faster using the **MsgPack** approach described above for certain types of data structures, such as lists. However, for most other types of data structures, native R serialization is usually faster. Most of the time-saving comes from the fact that the serialized **MsgPack** message is considerably smaller, leading to faster compresion time and/or faster reading and writing to disk.

Summary

The examples given above show how working with the R interface functions the C++ headers in **RcppMsgPack** can be useful in transferring data and integrated within data analysis workflows. The package allows fast and flexible serialization of generic data structures in R, and equivalent de-serialization of objects from other langauages. The **RcppMsgPack** package is hosted on CRAN, and can be installed via the standard `install.packages("RcppMsgPack")` command.

Bibliography

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016. [p6]

Apache Arrow Developers. Apache Arrow: A cross-language development platform for in-memory data. https://arrow.apache.org/, 2018. [p5]

T. Ching, D. Eddelbuettel, and the authors and contributors of MsgPack. *RcppMsgPack: 'MsgPack' C++ Header Files and Interface Functions for R*, 2018. URL https://CRAN.R-project.org/package=RcppMsgPack. R package version 0.2.2. [p1]

T. Dawborn and J. R. Curran. docrep: A lightweight and efficient document representation framework. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 762–771, 2014. [p1]

M. Dowle, A. Srinivasan, J. Gorecki, M. Chirico, P. Stetsenko, T. Short, S. Lianoglou, E. Antonyan, M. Bonsch, and H. Parsonage. *data.table: Extension of 'data.frame'*, 2018. URL https://cran.r-project.org/web/packages/data.table/index.html. R package version 1.11.4. [p6]

D. Eddelbuettel. *Seamless R and C++ Integration with Rcpp*. Springer, New York, 2013. ISBN 978-1-4614-6867-7. [p1]

D. Eddelbuettel. *RcppRedis: Rcpp Bindings for Redis using the hiredis Library*, 2017. URL https://cran.r-project.org/web/packages/RcppRedis/index.html. R package version 0.1.8. [p1]

D. Eddelbuettel and L. Silvestri. *nanotime: Nanosecond-Resolution Time for R*, 2017. URL https://CRAN.R-project.org/package=nanotime. R package version 0.2.0. [p2]

D. Eddelbuettel, M. Stokely, and J. Ooms. RProtoBuf: Efficient cross-language data serialization in R. *Journal of Statistical Software*, 71(2):1–24, 2016. doi: 10.18637/jss.v071.i02. [p1]

D. Eddelbuettel, R. François, J. Allaire, K. Ushey, Q. Kou, N. Russel, J. Chambers, and D. Bates. *Rcpp: Seamless R and C++ Integration*, 2018. URL http://CRAN.R-Project.org/package=Rcpp. R package version 0.12.17. [p1]

S. Furuhashi. MessagePack. https://msgpack.org//, 2018. [p1]

J.-L. Gailly and M. Adler. Zlib compression library, 2018. URL https://zlib.net/. [p7]

Google. Protocol Buffers. http://code.google.com/apis/protocolbuffers/, 2018. [p1]

S. T.-B. Hamida, E. B. Hamida, and B. Ahmed. A new mhealth communication framework for use in wearable wbans and mobile technologies. *Sensors*, 15(2):3379–3408, 2015. [p1]

IEEE Standards Committee. 754-2008 IEEE standard for floating-point arithmetic. *IEEE Computer Society Std*, 2008, 2008. [p2]

A. Krizhevsky. Learning multiple layers of features from tiny images. 2009. [p5]

Y. LeCun, C. Cortes, and C. Burges. Mnist handwritten digit database. *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2, 2010. [p5]

MongoDB. Bson (binary json) serialization, 2018. [p1]

J. Ooms. The jsonlite package: A practical and consistent mapping between json data and r objects. *arXiv:1403.2805 [stat.CO]*, 2014. URL http://arxiv.org/abs/1403.2805. [p1]

R. Srinivasan. *XDR: External data representation standard. No. RFC 1832*, 1995. URL http://www.rfc-editor.org/rfc/pdfrfc/rfc1832.txt.pdf. [p7]

S. v. d. Walt, S. C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011. [p6]

H. Wickham. *httr: Tools for Working with URLs and HTTP*, 2017. URL https://CRAN.R-project.org/package=httr. R package version 1.3.1. [p5]

H. Wickham, RStudio, Feather developers, Google, and LevelDB Authors. *feather: R Bindings to the Feather API*, 2016. URL https://CRAN.R-project.org/package=feather. R package version 0.3.1. [p5]

Travers Ching
Unaffiliated
ORCiD: 0000-0002-5577-3516
traversc@gmail.com

Dirk Eddelbuettel
Department of Statistics
University of Illinois at Urbana-Champaign
725 S Wright St
Champaign, IL 61820
ORCiD: 0000-0001-6419-907X
dirk@eddelbuettel.com