Proseminar

Programmieren in R

Rcpp

Oliver Heidmann
Betreuer: Julian Kunkel

Deutsches Klima Rechenzentrum
University of Hamburg

30.September 2016

Contents

Abstract

In most cases R is fast enough but there are times when more speed is needed than R offers. Rcpp allows the usage of C++ code in R scripts. And through this gain part of the performance C++ offers. In this report I will present Rcpp and explain its usage and document the speedups gained through it. This report is written with the intend to get C++ programmers into R and Rcpp so there will only be a very brief section about C++. At the end of the report the reader should be able to use Rcpp and to build his own Rcpp package without having previous knowledge about R or Rcpp.

1 R

R is a platform independent interpreted open source programming language. It is a implementation of S and combines S with lexical scoping semantics and also is heavily inspired by Scheme. Freeing of allocated memory space is done by a garbage collector. R is not made for performance but to offer tools to easy implement data analysis and statistics programs and use of many data sources eg. ODBC-compliant sources and other statistical packages. Comprehensive R Archive Network (CRAN) offers multitudes of free extensions called packages.

2 C++

C++ is a programming language designed for flexibility in use. In encompasses high level language features as well as access to the lower features. C++ allows a variety of different programming paradigms including Object-oriented programming. It allows direct control of memory management and is designed to have high performance in execution and memory usage. C++ code needs to get compiled and the language is strongly typed.

3 Basics

3.1 Installing packages

Installing packages is done through the

```
install.packages("pkgname","repo_url")
```

command. In an active R shell, if the user does not specify a mirror, the shell will ask the user to select one. For this report I wrote a small example script

```
requiredPackages = c(
    "Rcpp",
    "microbenchmark",
    "ggplot2")

#set repo url
repo <- "http://cran.uni-muenster.de/"

# for all required
# packages do:
for(p in requiredPackages)
{
    #if the package is not installed
    if(!require(p, character.only = TRUE))
    {
        #install the missing package
        install.packages(p, repo)
    }
}
```

which checks if all required packages are installed and installs the missing ones.

3.2 RObject

RObjects are taking the role of the base of all API classes in Rcpp. They themselves have no meaning and are used to bind the four parent classes together. SlotProxyPolicy is one of the parent classes and defines the member functions to manipulate the S objects which are used in R. The second is AttributeProxyPolicy and defines member fucntions that enable working with object attributes. RObjectMethods on the other hand define the functions of all RObject API classes. Preserve Storage is the last of the parent classes and provides the data field of the underlying R structure of type SEXP. In addition one of the most important attributes of RObject is that is protects the R variables used in the C++ code from the R garbage collector.

3.3 SEXP

SEXP is short for S-Expressions and is the type R uses and which Rcpp matches to C++ Objects.

3.4 Type Mappings

Rccp can map anything that offers the SEXP() method from C++ to R. SEXP() will be used by wrap to create the new SEXP object.

```
template <typename T>
    SEXP wrap(const T &  object)
```

And for R to C++ there is

```
template <typename T>
    T as(SEXP x)
```

The programmer can write his own SEXP() for user defined Types. Rcpp provides mappings for, int, double, bool, std::string and all standard library containers containing the just mentioned types. There are things that can not be mapped automatically for example a vector in a vector. For this Rcpp provides special types for the given example there is Rcpp::NumericMatrix which will be used later in this article.

4 Integrating C++ code

Rcpp offers different approaches of integrating C++ code into R. In the following section I will present them. Each of them differ in ease of use, performance and the amount of steps needed to use them.

4.1 Inline

Integrating C++ function with this method is best used for short C++ code. Inline allows the programmer to write C++ functions directly in the R file. To use inline the inline package must be included as well as the Rcpp package. After the function is defined it can be used as one would use a R function.

```
#including the packages
require(inline)
require(Rcpp)
```

```
#binding function to
#name
inlineMean <-cppFunction('
#c++ function
float sCpp_mean(
    #this function calculates
    #the mean of a vector
    std::vector<int> vec)
    {
        float result = 0;
        for(float entry :  vec)
        {
            result += entry;
        }
        return result/vec.size();
    }
')
```

In this example a function inlineMean was created through cppFunction(). Inside the cppFunction's parameter the C++ function is defined. The created function then then be called like any other R function. Each time the R interpreter reaches a inline c++ function it is compiled. Since compilation is relatively slow this burdens the execution time of the script. In addition to the example above I implemented two other functions. Both will be used in the benchmark section. The first takes a vector and calculates the mean.

```
inline_search <-cppFunction('
bool inline_search (
    std::vector<int> vec)
{
  bool found = true;
  unsinged long i;
  for (i = 0; i < vec.size(); i++) {
// if found iterator to the object
// else iterator to end of vec
    auto iter = std::find(vec.begin(),
                          vec.end(),
                          i);
    if (it == vec.end())
    {
        found = false;
    };
  }
    return found;
}
')
```

The second multiplies two matrices

```
inline_matrix_mul <-cppFunction('
Rcpp::NumericMatrix sCpp_matrix_mul(
    Rcpp::NumericMatrix a,
    Rcpp::NumericMatrix b)
    {
    unsigned long n = a.nrow();
    unsigned long m = a.ncol();
    unsigned long l = b.ncol();

    Rcpp::NumericMatrix result;

    if (a.ncol() == b.nrow()) {
      for (unsigned long i = 0; i < n; i++)
        for (unsigned long k = 0; k < l; k++)
          for (unsigned long j = 0; j < m; j++)
          {
            result(i,j) += a(i,k) * b(k,j);
          }
    } else {
        std::cout << "Error: matrices dont
                      have the right
```

```
                    dimensions <<
                  << std::endl;
        exit(EXIT_FAILURE);
    }
    return result;
  }
')
```

The time needed to compile them is shown here. All results are in seconds.

```
compiling inline_search
Compiling time inline search:
   user  system elapsed
  5.373   0.235   5.630

compiling inline_mean
Compiling time inline mean:
   user  system elapsed
  8.046   0.312   8.395

compiling inline_matrix_mul
Compiling time inline_matrix_mul:
   user  system elapsed
 10.659   0.423  11.131

Summation of inline compile time:
   user  system elapsed
 24.078   0.970  25.156
```

4.2 cppSource

Rccp offers the cppSource("filename") function to include entire Cpp source files. To use the functions in that file each function that is meant to be executed in the R file has to be marked with

```
//[[Rcpp::export]]
```

These function must be in the global namespace and parameters and return values need to be Mappable. Functions can be renamed with he extension of the export tag, this allows to use names which C++ would not allow.

```
//[[Rcpp::export(name = ".newname")]]
```

With this it is possible to, for example, adapt the name to your, in R used, function name conventions. For dependencies to other R packages there is the

```
//[[Rcpp::depends()]]
```

feature. This causes the cppFunction() function to configure the building process to compile and link to given packages. Mapping the types is done while the R code is interpreted so no manual mapping is required. As long as the C++ source file is not changed the compilation is only done once per R session. But every time the script is run through Rscript the C++ functions will be compiled again as this counts as a new R shell. The time needed to compile depends on the code that is compiled. The time needed to include functions very similar to the example fuctions from the inline section in second is shown here:

```
Loading cppSource source file
Loading time
```

```
   user  system elapsed
  2.753   0.139   2.902
```

included C++ headers will be linked and added to the source code automatically.

4.3 Package

For when there are lager c++ code parts and functionality that can be used in different contexts there is a simple way of creating packages which use Rcpp. Rccp offers a command that builds a skeleton package which also can be used as a tutorial/starting point to get into Rcpp and package creation.

```
> Rcpp.package.skeleton("pakagename")
```

Trough using the skeleton the package already confirms with the Rcpp vignette guideline which I will not go over in this report. The skeleton includes a file structure as well as example functions and readme files as well as skeleton package description/documentation files. The structure is separated into a usage folder and R, C++ code folders as well as a folder for the interface functions written in C++.

```
simplePackage/
  man/
  R/
  src/
  DESCRIPTION
  NAMESPACE
  Read-and-delete-me
```

The C++ Code is compiled into a shared library when the package is build so no further compilation is needed when executing the R script.

4.4 Rcpp.package.skeleton usage

Each C++ function requires a wrapper function, defined in the src folder, which cares for mapping the types between C++ and R. Rcpp offers the function compileAttributes which generates the wrappers and the bindings. For easy use I wrote a 2 line script compileAttributes.R, which takes care of generating the export functions.

```
>library(Rcpp)
>compileAttributes(commandArgs(TRUE))
```

The script can be called with with

```
Rscript compileAttributes.R <pkg_name>
```

without the need to open an active R shell.

The compileAttributes function would then generate the following code from a C++ function.

- C++ function:

```
// [[Rcpp::export]]
std::vector<int> add(
    int a, int b)
```

```
{
    return a + b;
}
```

- the generated wrapper function

```
using namespace Rcpp;
using namespace std;

//function definiton
int add(int a, int b);

//wraper function
RcppExport SEXP
test_add(SEXP aSEXP, SEXP bSEXP)
{
  BEGIN_RCPP
    RObject rcpp_result_gen;
    RNGScope rcpp_rngScope_gen;
    traits::input_parameter<int>::type a(aSEXP);
    traits::input_parameter<int>::type b(bSEXP);
    rcpp_result_gen = wrap(add(a, b));
    return rcpp_result_gen;
  END_RCPP
}
```

- function binding through .call

```
> r_function <- function(p1_, p_2, p_n){
    .Call('cppfunc_name'),
    PACKAGE = 'pkg_name',
    p1_, p_2, p_n}
```

The .Call method is the interface function between R and C++.

4.5 Installing the package

installing the package is done through

```
R CMD INSTALL --build test
```

once installed it can be used like any other package.

5 Performance

I used the above introduced functions to compare inline, sourceCpp and the package in terms of running time. In addition I compared the results with a native C++ program and a native R program. Those three function each represent an often needed task in programming

1. Searching entries in a vector
2. Matrix multiplication
3. Calculations on a vector

For the benchmarks I used the package microbenchmarks.

5.1 System data

System Specs:

1. Architecture: x86 64
2. OS: linux 4.7.4-1 (ArchLinux)
3. Ram: 8GB DDR3
4. Cpu: Intel(R) Core(TM) i5-3570K CPU @ 3.40 GHz

5.2 Inline vs sourceCpp vs package

All functions in a group differ only in their way of calling them from R. Each function was called 100 times. The source code is attached to this report for full view of the functions and the benchmarks. Important to note is that for the compilation of the package the compiler optimization flag O2 was used.

5.2.1 Mean calculation

This function goes over a vector and sums its entries up. After that the function divides the sum by the number of elements in that vector.

	min(ms)	mean (ms)	max(ms)
sCpp	0.028000	0.03778730	0.071649
inline	0.028376	0.03791835	0.095817
package	0.030758	0.04214210	0.141447

5.2.2 matrix multiplication

In the matrix multiplication functions I used no special C++ features. It is implemented trough three nested for loops. The loops are nested in a way that minimizes cache misses in case of larger matrices.

This function works by going over the vector once while calculating the sum of all contained values. At the end the sum gets divided by the size of the vector.

	min (ms)	mean (ms)	max (ms)
sCpp	1.316425	1.378768	1.495590
inline	1.320563	1.389122	1.611258
package	1.319452	1.389438	1.584307

5.2.3 Search

For the search I used the std::find(...) function from the C++ Standard Library. The vectors in which the algorithm searches are filled with ordered numbers and each number is searched for once.

	min (ms)	mean (ms)	max (ms)
sCpp	3334.03500	34.68003	36.38635
inline	3334.07672	34.74363	36.43009
package	3334.12335	34.67028	36.14496

5.2.4 Results

The results show that each of the here presented usages of Rcpp are quite close together in terms of performance. Using sourceCpp is just slightly faster that the other two as long as we do not add in the compile times for cppSource and inline. If we add those in the package is the fastest of the three.

5.3 C++ vs R vs Rcpp

In this section I use the package used in the benchmarks before and compare the runtime to the native R and C++ implementations.

	matrix	mean	search
R	1583.8 ms	2.00 ms	269.51 ms
Rcpp	1.37 ms	0.043 ms	33.83 ms
C++O2	1.4 ms	0.02 ms	26.43 ms
C++O3	0.24 ms	0.01 ms	17.67 ms

5.3.1 Result

The tables in these section show the decrease of runtime though using Rcpp and also the difference to native C++ code. They also show the further potential decrease in runtime by changing the by R/Rcpp used O2 compiler flag to O3. O3 tells the compiler to optimize the code as much as

it is possible by the compiler.
For the matrix Benchmarks we get the following speed up.

	R	Rcpp
Rcpp	1154.36	0
C++ O2	1130.62	0.98
C++ O3	6595.3	5.70

There is a very huge speed from native R code to C++. The table also shows that through using the C++ compilers optimization there is even more room for improvement. In fact the difference between O2 and O3 is a six times speed up in addition to the 1000 times speed up from native R to Rcpp.

In the mean we get a lot less reduction in runtime but these results also show a huge difference in runtime through Rcpp.

	R	Rcpp
Rcpp	46.51	0
C++ O2	100	2.15
C++ O3	200	4.3

Using the O3 flag would half the runtime in addition to the 46 times lower runtime of Rcpp.

The searches show the lowest decrease of runtime.

	R	Rcpp
Rcpp	7.967	0
C++O2	10.20	1.27
C++O3	15.25	1.91

Here we get a nearly 8 times decrease in runtime through using Rcpp. And almost double that for a C++ program compiled with the O3 flag.

6 Conclusion

In the first part I showed that R/Rcpp are easy to get into. Many already integrated tools like the package creation or the R function which creates mapping and wrapper functions for the programmer make working with R/Rcpp very easy and comfortable.

Through using Rccp the resulting speed up is huge, especially for loops, which are generally slow in R profit from the C++ integration. But not only loops get a huge boost but every function tested got at least 7 times faster that its native R counter part. In addition to the already speed up extra optimization is possible through changing the compiler flag R/Rcpp uses for compiling C++ code to O3.

Using inline is good for short code snippets as it is easily written in. For lager functions or multiple large functions it is less appropriate since each function is compiled when reached the first time. Also it decreases readability and thus gives the programmer a harder time maintaining the code.

For mid sized C++ code cppSource ist the best way to integrate the C++ functions. It has similar ease of use as inline and keeps, through the way the C++ code is included, R and C++ code in separate files. Another benefit is that the C++ code is easily reusable since all that has to be done to use it, is to include it with sourceCpp. As inline cppSource has the drawback of the needed compile time when executing the Rscript.

Using a package that contains the C++ offers the most increase in performance since the package does not need to compile the code each time it is used as long as it has already been installed. I is also the best way to maintain a large C++ code base. One drawback is that it need the most amount of work to be implemented as such it is not a good choice for prototyping or small code snippets. As the other two do it does handle the wrapping an mapping of functions and types on its own as long as the compileAttributes function is used. Its file structure, generated through Rcpp.package.skeleton, makes the code easy to maintain and extend. Through this, keeping the readability of the package high is also made easy.

All in all is Rcpp a very good and easy way to increase the performance of Rscripts and offers many tools to ease the implementation process for the programmer.

- adv-r.had.co.nz/Performance.html
  29.9.2016
- http://dirk.eddelbuettel.com/code/rcpp/Rcpp-introduction.pdf
  13.6.2016
- http://dirk.eddelbuettel.com/code/rcpp/Rcpp-package.pdf
  13.6.2016
- http://dirk.eddelbuettel.com/code/rcpp/Rcpp-FAQ.pdf
  13.6.2016
- https:://www.techopedia.com/definition/26184/c-programming-language
  13.6.2016
- adv-r.had.co.nz/Rcpp.html
  13.6.2016
- https://cran.r-project.org/web/packages/microbenchmark/microbenchmark.pdf
  13.6.2016