HPC-R Exercises: Interfacing to Compiled Code

Drew Schmidt

02/27/2015

Interfacing to Compiled Code

Rcpp

1. Write a C++ function callable by R that takes an integer value and multiplies it by 2. What happens when you pass this function a double (like 3.2)? A string (like "3")?
2. Write a C++ function callable by R that takes an `Rcpp::NumericVector` as an input, and returns the sum of its elements.
3. Write a C++ function callable by R called `anyNA()` that takes an `Rcpp::NumericVector` as an input, and returns `TRUE` if there is an `NA` among the vector's values, and `FALSE` otherwise. (Hint: use `ISNA()` to check).
4. Modify the function from exercise 2 to take an additional input named `na_rm` of type `bool` so that the function behaves like R's `sum()` function with `na.rm` argument.
5. Write a C++ function callable by R that takes an `Rcpp::NumericMatrix` as an input, and returns the vector of column sums. If you completed exercise 2 from the previous section, compare the performance of the two approaches.

Answers

Rcpp

1. One possibility is:

```
library(Rcpp)

code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
int my_times_two(int x)
{
  return x*2;
}
"

sourceCpp(code=code)
```

Observe that Rcpp, for better or worse, will cast inputs as needed:

```
my_times_two(3.2)
```

```
## [1] 6
```

or if not possible, throw an error:

```
tryCatch(my_times_two("3"), error=print)
```

```
## <Rcpp::not_compatible in eval(expr, envir, enclos): not compatible with requested type>
```

2. One possibility is:

```
library(Rcpp)

code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
double my_sum(Rcpp::NumericVector x)
{
  double sum = 0.;

  for (int i=0; i<x.size(); i++)
    sum += x[i];

  return sum;
}
"

sourceCpp(code=code)
```

```
my_sum(1:10) == 10*11/2
```

```
## [1] TRUE
```

3. One possible solution is:

```
library(Rcpp)

code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
bool anyNA(Rcpp::NumericVector x)
{
  bool ret;

  for (int i=0; i<x.size(); i++)
  {
    if (ISNA(x[i]))
```

```
      return true;
  }

  return false;
}
"

sourceCpp(code=code)
```

This naturally leads to a fairly amusing example:

```
library(rbenchmark)

x <- runif(1e8)
x[1] <- NA

benchmark(R=any(is.na(x)), CXX=anyNA(x), columns=c("test", "elapsed", "relative"))
```

```
##   test elapsed relative
## 2  CXX   0.001        1
## 1    R  34.493    34493
```

In the R version, first a logical vector is constructed, recording whether or not values of `x` are `NA`. In practice, performance gains from using C++ are much more modest.

4. One possible solution is:

```
library(Rcpp)

code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
double my_sum2(Rcpp::NumericVector x, bool na_rm)
{
  double sum = 0.;

  if (na_rm)
  {
    for (int i=0; i<x.size(); i++)
    {
      if (!ISNA(x[i]))
        sum += x[i];
    }
  }
  else
  {
    for (int i=0; i<x.size(); i++)
      sum += x[i];
  }

  return sum;
```

```
}
"

sourceCpp(code=code)
```

```
my_sum2(1:10, TRUE)
```

```
## [1] 55
```

```
my_sum2(1:10, FALSE)
```

```
## [1] 55
```

```
my_sum2(c(1, 2, NA, 4, 5), TRUE)
```

```
## [1] 12
```

```
my_sum2(c(1, 2, NA, 4, 5), FALSE)
```

```
## [1] NA
```

5. One possible solution is:

```
library(Rcpp)

code <- "
#include <Rcpp.h>

// [[Rcpp::export]]
Rcpp::NumericVector my_colSums(Rcpp::NumericMatrix x)
{
  Rcpp::NumericVector ret(x.ncol());

  for (int j=0; j<x.ncol(); j++)
  {
    ret[j] = 0.0;

    for (int i=0; i<x.nrow(); i++)
      ret[j] += x(i, j);
  }

  return ret;
}
"

sourceCpp(code=code)
```

We could have iterated over the rows first, then columns. However, as R matrices are column-major, you are advised to iterate over columns first whenever possible.

```r
x <- matrix(1:30, 10)

colSums(x)
```

```
## [1]  55 155 255
```

```r
my_colSums(x)
```

```
## [1]  55 155 255
```