High-Performance Computing with R
Interface for compiled code and paralle computation

Chao Cheng

School of Statistics and Management
Shanghai University of Finance and Economics, China

November 16, 2020

Outline

1. R with C/C++
   - Introduction
   - R with C
   - R with C++
2. Parallel Computation in R
   - Introduction
   - The Parallel package
   - Advanced topics
   - Easy parallel computation
   - Other stuff
3. Acknowledgement

Introduction

A toy example

- $1 + 2 + \cdots + 100$.
- Compute the sum of a vector.

Why is R/Python slow?

- Interpreted Languages vs. Complied Languages.
- Trade off: Extreme dynamism vs. Runtime speed.

Abstraction

We need an interface to communicate with foreigen languages from R.

Preparation

C Compiler

- Windows users: install Rtools. (Might need to set the `BINPREF` variable if you install Rtools to a custom location.)
- Mac users: check R's website for instructions.
- Linux users: Use your package manager or follow R's documentation.

Useful reference

- Advanced R(1st edition), the last chapter.
- Writing R Extentions(especially Chapter 5 and 6).
- Header files located at (`R.home("include")`).

API

R

- `R CMD SHLIB`, `dyn.load`, `dyn.unload`.
- `.Call`, `.External`, `.C`, `.Fortran`.
- `.Platform$dynlib.ext` checks the file extension.

C

- Header files: `R.h`, `Rinternals.h`, `Rmath.h`, etc.
- Basic type: `SEXP`. `REAL`, `coerceVector` etc.
- Memory and GC: `allocVector`, `PROTECT`, `UNPROTECT`.
- RNG: `GetRNGstate`, `PutRNGstate`, `rnorm`, etc.

- Simple examples at sum1.c and sum2.c.

A simple template

```
#include<WhatYouNeed.h>
/* ---some pre-define functions --- */
double function1(int x){
  ...
}
/* --- main function --- */
SEXP YourFunction(SEXP Arg_from_R, SEXP ...){
  SEXP x;
  x = PROTECT(allocXXX(XXXSXP, m));
  ...
  UNPROTECT(n);
  return(x);
}
```

Questions

API related

- What are the `XXXSXP`s for logical, character and factor vectors?
- What are the `allocXXX`s for a matrix, 3D-array, nD-array and list?
- What about `log1p()`, `log1pmax()`, `log1pexp()` and how many R functions I can use in C?

Numerical Problems

- Linear algebra
- Fast Fourier Transformation
- Root finding
- Derivative and integration
- ...

Possible Answers

API related

- Check the Useful Reference metioned earlier.

Numerical Problems

- Numerical Recipies(2nd Edition).

R with C++

The hard way

`extern "C"` + R's internal API.

- A simple demonstration using sum3.cpp.

The easy way

The `Rcpp` package.

API

R

- `Rcpp::sourceCpp()`.
- Do NOT use `inline` nor `cppFunction` nor `evalCpp` in your R script.

C++

- Header file: `include<Rcpp.h>`.
- Denote the function exported to R with `//[[Rcpp::export]]`.

- A simple demonstration using test.cpp.

Useful reference

Starting point

- Rcpp for everyone: `https://teuder.github.io/rcpp4everyone_en/`
- Advanced R, Chapter25, Rewriting R code in C++
- Rcpp vignettes: introduction, quickref, attributes and FAQ
- `RcppExample` package.

Advanced

- Rcpp vignettes: remains
- Seamless R and Cpp Integration with Rcpp
- Rcpp reference manual(3k+ pages)
- Source code

Data types

- See Rcpp for everyone for more details.

Value	R vector	Rcpp Vector	Rcpp Matrix
Logical	logical	LogicalVector	LogicalMatrix
Integer	integer	IntegerVector	IntegerMatrix
Real	numeric	NumericVector	NumericMatrix
Complex	complex	ComplexVector	ComplexMatrix
String	character	CharacterVector	CharacterMatrix

R	Rcpp
list	List
data.frame	List

Rcpp::List

Create a List

```
// Create list L from vector v1, v2
Rcpp::List L = Rcpp::List::create(v1, v2);

// When giving names to elements
Rcpp::List L = Rcpp::List::create(Named("name1") = v1 ,
                                  _["name2"] = v2);
```

Accessing List elements

```
NumericVector v1 = L[0];
NumericVector v2 = L["V1"];
```

Rcpp::Function

Accessing R functions

```
// calling rnorm()
Rcpp::Function myfun("rnorm");

// This code is interpreted as rnorm(n=5, mean=10, sd=2)
myfun(5, Named("sd")=2, _["mean"]=10);

// return type from Rcpp::Function is SEXP
Rcpp::NumericVector res = Rcpp::as<Rcpp::NumericVector>(myfun(10))
```

As and Wrap

```cpp
// conversion from R to C++
template <typename T> T as(SEXP m_sexp);
// conversion from C++ to R
template <typename T> SEXP wrap(const T& object);
```

In practise

```cpp
Rcpp::as<Cpp_Typename>(SEXP_Object)
Rcpp::wrap(Cpp_Object)      // return the SEXP
```

Print message

Rcout and Rcerr

It's the same as `std::cout` and `std::cerr`.

```
// printing value of vector
Rcpp::Rcout << "The value of v : " << v << std::endl;

// printing error message
// Handled as R message, not stop the running programe
Rcpp::Rcerr << "Error message" << std::endl;
```

Rprintf() and REprintf()

It's the same as `printf()` and `Eprintf()`.

Personal suggestion

- Be explicit, use `::`, do NOT use `using namespace xxx`.
- Check for user interuption during long computation

```
for (int i=0; i<1000000; i++) {
// check for interrupt every 1000 iterations
if (i % 1000 == 0)
Rcpp::checkUserInterrupt();
// ...do some expensive work...
}
```

Extention of Rcpp

Great applications of Rcpp

- Linear algebra: RcppArmadillo and RcppEigen.
- Scientific computation: RcppGSL.

API

```
#include<RcppEigen.h>
//[[Rcpp::depends(RcppEigen)]]
```

- Refer to the documentation of Armadillo, Eigen and GSL for more details.

Extention of Rcpp

Dirk's blog and Rcpp Gallery

- `tidyCpp` An C++ layer on top of the C API for R.
- `RcppParallel`
- Call Python from R through Rcpp.
- `RcppProgress` is a tool to help you monitor the execution time of your C++ code.
- ......

Extention of Rcpp

RcppProgress

- Article: Using RcppProgress to control the long computations in C++

```
# Rcpp::sourceCpp("rcpp_progress.cpp")
# long_computation(3000)
# 0%   10   20   30   40   50   60   70   80   90   100%
# [----|----|----|----|----|----|----|----|----|----|
# **************************************************|
# [1] 3002.32
```

Extention of Rcpp

RcppParallel

```cpp
#include<Rcpp.h>
// [[Rcpp::depends(RcppParallel)]]
#include <RcppParallel.h>

struct My_Sum : RcppParallel::Worker{
    // member variables
    const RcppParallel::RVector<double> input;
    double res;

    // constructor
    My_Sum(Rcpp::NumericVector x) : input(x), res(0.0) {}
    My_Sum(const My_Sum& my_sum, RcppParallel::Split): input(my_sum.input), res(0.0) {}
    // operator functions
    void operator()(std::size_t start, std::size_t end){
        res += std::accumulate(input.begin() + start,
                               input.begin() + end,
                               0.0);
    }
    void join(const My_Sum& rhs){
        res += rhs.res;
    }
};
// [[Rcpp::export]]
double par_sum(Rcpp::NumericVector invec){
    My_Sum my_sum(invec);
    RcppParallel::parallelReduce(0, invec.size(), my_sum);
    return(my_sum.res);
}
```

```r
> library(rbenchmark)
> library(Rcpp)
> sourceCpp("rcpp_par2.cpp")
> x <- as.numeric(1 : 1000000)
> res1 <- sum(x)
> res2 <- par_sum(x)
> res3 <- rcpp_sum(x)
> identical(res1, res2)
[1] TRUE
> identical(res2, res3)
[1] TRUE
> benchmark(sum(x), par_sum(x), rcpp_sum(x), order = "relative")[, 1 : 4]
         test replications elapsed relative
2  par_sum(x)          100    0.03    1.000
1      sum(x)          100    0.11    3.667
3 rcpp_sum(x)          100    0.11    3.667
>
```

reticulate, R interface to Python

R

```r
library(reticulate)
use_condaenv("base")
os <- import("os")
os$listdir()
source_python("flights.py")
flights <- read_flights("flights.csv")
```

Python script

```python
import pandas
def read_flights(file):
  flights = pandas.read_csv(file)
  flights = flights[flights['dest'] == "ORD"]
  flights = flights[['carrier', 'dep_delay', 'arr_delay']]
  flights = flights.dropna()
  return flights
```

Introduction

A toy example

- $1 + 2 + \cdots + 100$
- Compute the sum of a vector

Abstraction

- The whole job can be break into small parts and they can be done independently of each other.
- Map + Reduce

Useful cases

- Simulation
- Bootstrap and MCMC
- Elementwisely update an vector in ADMM algorithm

Basic paralle computation for simulation

- Start multiple R sessions
- Preparation: load necessary packages, etc.
- Run simulation scripts, possibly according to session ID.
- Collect and summary the results by hand.

Abstraction

- Create workers
- Prepare workers
- Run script in parallel and collect the results.

The `parallel` package

- It's derived from `snow` and `multicore` packages.
- Useful reference:
  - Parallel R. This book is a bit old.
  - `parallel`'s documentations.
  - `parallel`'s vignettes.

A simple template

```
library(parallel)
# use all the cores of this machine
cls <- makeCluster(detectCores())
# split the task index SEQ to workers
ind_seq <- clusterSplit(cls, SEQ)

# initializing workers
clusterEvalQ(cls, fun)
# pass VARLIST from master to all the workers
clusterExport(cls, VARLIST)

# Carry out the task parFUNCTION parallely
parLapply(cls, ind_seq, parFUNCTION)

# stop workers
stopCluster(cls)
```

A simple example

```
library(parallel)
a <- rnorm(12)
slow_function <- function(invec){
    ...     # a slow function
}

cls <- makeCluster(4)
ind_seq <- clusterSplit(cls, a)
clusterExport(cls, varlist = "slow_function")
res_par <- parSapply(cls, ind_seq, slow_function)
res <- sum(res_par)
```

PSOCK vs FORK

```r
a <- rnorm(100)
```

PSOCK

```r
cls <- makeCluster(4)      # default type is PSOCK
```

FORK

```r
cls <- makeCluster(4, type = "FORK")     # NOT available on Windows
```

```r
parSapply(cls, 1 : 10, function(id){
    return(a[id])
})
```

PSOCK vs FORK

PSOCK

- Pros:
  - Use socket connection, a general approach.
  - All system, locally or remotely with suitable setup such as MPI
- Cons:
  - Might be hard to configure.
  - Manually transport the data.

FORK

- Pros: use FORK mechanism, no worry about variable transportation.
- Cons: Only for one machine, not available on Windows.

Parallel random number generation

- Manually use `set.seed()` on every worker.
- Use 'L'Ecuyer-CMRG' multiple RNG stream.
  1. `RNGkind("L'Ecuyer-CMRG")` on your main session.
  2. `set.seed()` on your main session.
  3. `clusterSetupRNGstream()` to set your workers' seed.

Variable transportation

- Explicit functions and variables will always be transported.
- FORK will copy the main session at creation.
- Others should be taken care of by hand.
- Additional configuration of `clusterExport` when nested in a function call.

Dark time of parallel computation

There are so many different parallel backends:

- snow
- multicore
- parallel
- MPI
- Redis
- Hadoop
- Spark
- Slurm
- ...

How to support them? How to maintain code?

Foreach

`foreach` defines a simple but powerful framework for map/reduce parallel computation.

Package author/code writer

Decide which part of code can run in parallel.

End user

Decide how to run in parallel based on their available resources.

`foreach` is syntactically structured in the form of a for loop.

Foreach

```r
library(foreach)
# library(doParallel)
# registerDoParallel()
a <- 10
foreach(i = 1 : 12, j = 12 : 1, .combine = rbind) %dopar%{
    Sys.sleep(0.5)
    print(paste("i = ", i, ", j = ", j, sep = ""))
    data.frame(i, j, a)
}
```

future and future.apply

- `future` provides a simple and uniform way of evaluating R expressions asynchronously using various resources available to the user.
- `future.apply` provides worry-free parallel alternatives to base-R "apply" functions.

```
library(future.apply)      # default plan is sequential
# plan(cluster)
x <- rnorm(16)
future_lapply(1 : 5, function(id){
    print(paste("id = ", id, sep = ""))      # normal print kept
    Sys.sleep(0.5)
    sum(x[1 : id])
})
```

future

- Asynchronous computation. Not constrained by a for-loop or apply syntax.
- Available extensions:
  - `future.apply`
  - `doFuture`: backends for `foreach`, `BiocParallel` and `plyr`.
  - `furrr`

future

```r
library(future)
plan(cluster)

x <- future({
    x <- matrix(rnorm(10 ^ 6), nrow = 10 ^ 3)
    for(i in 1 : 5){
        print(paste("i = ", i))
        res <- eigen(x)
    }
    return(res)
}, seed = T)     # not block the main session

resolved(x)     # check whether the future is resolved
a <- rnorm(10)     # we can do other stuff at the main session
```

Personal suggestions

- Nested parallel is NOT recommended. At least it should be done with careful configuration.
- `future.apply` vs `foreach`
  - Familiar with `foreach`: just use the `doFuture` backends.
  - New to parallel: `future.apply` is a good start point for your code.
  - `future` backend will relay the printed messages.
  - Performance in parallel are close, so-called.
  - Performance for sequential are slower than for-loop.

I want progress bars

- `RcppProgress` allows to display a progress bar in the R console for long running computations taking place in c++ code, supports OpenMP.
- `pbapply` is a lightweight package that adds progress bar to vectorized R functions ('*apply'). It supports several parallel backends.
- `progress` shows ASCII progress bars.
- `progressr` provides a minimal API for reporting progress updates in R.
  - Developer is responsible for providing progress updates.
  - End user decides if, when, and how progress should be presented.

progressr

```r
library(progressr)
slow_sum <- function(x) {
    p <- progressr::progressor(along = x)
    sum <- 0
    for (kk in seq_along(x)) {
        Sys.sleep(0.5)
        sum <- sum + x[kk]
        p(message = sprintf("Added %g", x[kk]))
    }
    sum
}
# handlers("default")    # default handler is "txtprogressbar"
with_progress(y <- slow_sum(1:10))
handlers("progress")
with_progress(y <- slow_sum(1:10))
```

Acknowledgement

Thank you all for your attention!