Portfollio 9 - RcppParallel

Sam Bowyer

2023-06-04

In this portofolio we will discuss how the `R`-to-`C++` gains obtained in regular `Rcpp` can be improved upon further by parallelisation with `RcppParallel`. In particular, we'll look into `parallelFor` and `parallelReduce`.

parallelFor

In `RcppParallel`, the `parallelFor` loop takes in a start index, end index and a Worker object. The indices are used to split the loop into chunks, which are then assigned to threads, whilst the Worker object is used to perform the actual computation. The worker object must have a `operator()` method, which takes in a range of indices and performs the computation on them; the `parallelFor` loop then calls this method on each chunk of indices. A simple `parallelFor` loop will look like this, in particular below we're squaring each element of a vector:

```
library(Rcpp)
library(RcppParallel)
```

```
##
## Attaching package: 'RcppParallel'

## The following object is masked from 'package:Rcpp':
##
##     LdFlags
```

```
sourceCpp(code = '
#include <Rcpp.h>
#include <RcppParallel.h>
using namespace RcppParallel;

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

struct WorkerExample : public Worker{
  // Input vectors
  const RVector<double> input;

  // Output vector
  RVector<double> output;

  // Constructor
  WorkerExample(const Rcpp::NumericVector input, Rcpp::NumericVector output)
    : input(input), output(output) {}

  // Overloaded operator()
  void operator()(std::size_t begin, std::size_t end){
    for(std::size_t i = begin; i < end; i++){
      output[i] = input[i] * input[i];
```

```
    }
  }
};

// [[Rcpp::export]]
Rcpp::NumericVector parRcppSquare(Rcpp::NumericVector x){
  // Allocate the output vector
  Rcpp::NumericVector output(x.size());

  WorkerExample obj(x, output);

  // Square the elements with parallelFor
  parallelFor(0, x.size(), obj);

  // Return the output vector
  return output;
}
')
```

We'll compare this to the regular `Rcpp` implementation of the same function, both using a regular `for` loop and with a vectorised implementation.

```
sourceCpp(code = '
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
NumericVector vecRcppSquare(NumericVector x){
  return x * x;
}

// [[Rcpp::export]]
NumericVector RcppSquare(NumericVector x){
  // Allocate the output vector
  NumericVector output(x.size());

  // Square the elements
  for(int i = 0; i < x.size(); i++){
    output[i] = x[i] * x[i];
  }

  // Return the output vector
  return output;
}
')
```

As well as implementations in pure `R`, one with a vectorised approach and one with a regular `for` loop:

```
vecRSquare <- function(x){
  return(x^2)
}

RSquare <- function(x){
  output <- rep(NA, length(x))
  for(i in 1:length(x)){
```

```r
    output[i] <- x[i]^2
  }
  return(output)
}
```

Running these function on a vector of 1 million elements, we get the following results:

```r
library(microbenchmark)
x <- rnorm(1e6)
microbenchmark(R = RSquare(x), vecR = vecRSquare(x), Rcpp = RcppSquare(x),
               vecRcpp = vecRcppSquare(x), parRcpp = parRcppSquare(x))
```

```
## Unit: microseconds
##     expr       min         lq      mean    median        uq       max neval
##        R 27756.041 28545.8475 29762.857 29834.531 30507.714 35217.681   100
##     vecR   645.584   829.8040  1802.348   992.363  2199.082 32800.435   100
##     Rcpp  1588.169  1794.2670  2716.646  2867.247  3060.106  6385.440   100
##  vecRcpp   645.073   880.6485  1995.754  1903.530  2231.368  9584.592   100
##  parRcpp   886.353  1043.9585  1811.081  1391.324  2261.725  5734.928   100
```

Looking at the mean runtime column, we see that clearly the non-vectorised pure `R` implementation is by far the slowest, with the fastest implementation being the vectorised pure `R` one, closely followed by our `parallelRcpp` implementation. This table of results is interesting as it allows us to see the impact of computational overheads involved with each approach—in particular we can see that using a non-parallel, non-vectorised `Rcpp` implementation is very inneficient, being the second slowest in the above experiment. Clearly the benefits of each implementation-type depends on the problem at hand and the computation required therein—but for this simple squaring problem a vectorised `R` approach or `parallelRcpp` approach seem to be roughly jointly superior (with a very slight advantage obtained with the vectorised `R`).

parallelReduce

As discussed in Portfolio 7 on Intel TBB, another very useful function that can easily be parallelised is `reduce`, which is implemented in `RcppParallel` as `parallelReduce`. This works similarly to `parallelFor`, in that it takes in a start index, end index and a Worker object, but the Worker object must have an additional `join` method, which is used to combine the results of the parallel computation. We'll look at an example of this, based on [1] in which we'll compute the inner product of two vectors.

The serial `Rcpp` implementation of this function is as follows:

```r
sourceCpp(code = '
#include <Rcpp.h>
using namespace Rcpp;

// [[Rcpp::export]]
double RcppInnerProduct(NumericVector x, NumericVector y){
  double product = 0;
  for(int i = 0; i < x.size(); i++){
    product += x[i] * y[i];
  }
  return product;
}
')
```

We'll compare this to the `RcppParallel` implementation:

```r
sourceCpp(code = '
#include <Rcpp.h>
```

```
#include <RcppParallel.h>
using namespace RcppParallel;

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

struct InnerProduct : public Worker{
  // Input vectors
  const RVector<double> x;
  const RVector<double> y;

  // Output vector
  double product;

  // Constructors
  InnerProduct(const Rcpp::NumericVector x, const Rcpp::NumericVector y)
    : x(x), y(y), product(0) {}
  InnerProduct(const InnerProduct& obj, Split)
    : x(obj.x), y(obj.y), product(0) {}

  // Overloaded operator()
  void operator()(std::size_t begin, std::size_t end){
    double temp = 0;
    for(std::size_t i = begin; i < end; i++){
      temp += x[i] * y[i];
    }
    product += temp;
  }

  // Join method
  void join(const InnerProduct& rhs){
    product += rhs.product;
  }
};

// [[Rcpp::export]]
double parRcppInnerProduct(Rcpp::NumericVector x, Rcpp::NumericVector y){
  InnerProduct obj(x, y);

  // Square the elements with parallelFor
  parallelReduce(0, x.size(), obj);

  // Return the output vector
  return obj.product;
}
')
```

Note that the `join` method is used to combine the results of the parallel computation, in this case the `product` variable, and also that we have two constructors, one of which is used to split the computation into chunks. We'll compare these implementations to three pure `R` implementation, one using the `%*%` operator, one using a regular `for` loop and one using a vectorised approach:

```r
RInnerProduct <- function(x, y){
  product <- 0
  for(i in 1:length(x)){
```

```r
    product <- product + x[i] * y[i]
  }
  return(product)
}

vecRInnerProduct <- function(x, y){
  return(sum(x * y))
}

operatorRInnerProduct <- function(x, y){
  return(x %*% y)
}
```

Running these functions on two vectors of 1 million elements, we get the following results:

```r
x <- rnorm(1e6)
y <- rnorm(1e6)

microbenchmark(R = RInnerProduct(x, y), vecR = vecRInnerProduct(x, y),
               opR = operatorRInnerProduct(x, y), Rcpp = RcppInnerProduct(x, y),
               parRcpp = parRcppInnerProduct(x, y))
```

```
## Unit: microseconds
##     expr       min         lq       mean     median        uq       max neval
##        R 24554.145 25209.6670 25889.1679 25606.6630 26207.521 31554.197   100
##     vecR  1726.131  3003.1510  3436.2685  3217.5665  3730.384  5335.938   100
##      opR  2160.190  2425.3430  2679.5718  2585.0295  2869.661  3962.915   100
##     Rcpp  1681.060  1811.7020  1945.9117  1903.8730  2037.503  2467.551   100
##  parRcpp   138.091   285.1465   405.8396   386.7485   502.258  1196.881   100
```

Again, by looking at the mean runtime column, we see that the pure `R` implementations are by far the slowest, particularly the regular `for` loop implementation, whilst the `%*%` operator is the fastest of these three. But importantly, we see that the `RcppParallel` implementation is the fastest of all, with a speedup of around 6-7x over the `%*%` operator, and a speedup of around 5x over the regular `Rcpp` implementation. This is a very impressive speedup, and shows the power of parallelisation in `RcppParallel`.

References

[1] JJ Allaire. Computing an Inner Product with RcppParallel, July 2014.