Portfolio Report 10: Parallel Rcpp

Jake Spiteri

2020

In this report we investigate the use of parallelization in Rcpp. We will look at how to use the OpenMP API, and how to use the RcppParallel package.

OpenMP in Rcpp

To demonstrate the use of OpenMP in Rcpp, we construct a very simple example which simply sleeps for 1 second, and then prints "Hello world!".

```
library(Rcpp)
sourceCpp(code = '
#include <unistd.h>
#include <Rcpp.h>

// [[Rcpp::export(say_hello)]]
void say_hello(int nhello)
{
  for(int i=0; i<nhello; i++)
  {
    sleep(1);
    Rcpp::Rcout << "Hello World! \\n";
  }
}
')
```

We test the function below, and time it.

```
system.time(say_hello(2))
```

```
## Hello World!
## Hello World!
```

```
##    user  system elapsed
##       0       0       2
```

We now edit the above function to use OpenMP to parallelize the for loop. We will specify the number of cores to use as an argument to the function.

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

// [[Rcpp::plugins(openmp)]]
// [[Rcpp::export(say_hello_omp)]]
void say_hello_omp(int nhello, int ncores)
{

  #if defined(_OPENMP)
   #pragma omp parallel num_threads(ncores)
   #pragma omp for
  #endif
  for(int i=0; i<nhello; i++)
  {
    sleep(1);
    printf("Hello World! Printed by thread number %d\\n", omp_get_thread_num());
  }
}
')
```

In the above Rcpp function we have specified that we require the `openmp` plugin via `// [[Rcpp::plugins(openmp)]]`, specified the number of threads OpenMP should use via `#pragma omp parallel num_threads(ncores)`, and told the compiler that the for loop can be run in parallel via `#pragma omp for`. We have also included the OpenMP header file `omp.h` which gives us access to the `omp_get_thread_num()` function. Using this function we can also print which thread has printed which 'Hello World!'.

For illustrative purposes, we run the function `say_hello_omp` with `nhello`=5, and `ncores`=4.

```
say_hello_omp(5, 4)
```

As expected we see that one thread (most likely the 0th thread) runs the for loop twice. We can show the performance increase as the number of threads increases below.

```
speed_test
```

```
## Unit: seconds
##           expr       min        lq      mean    median        uq       max
##       one_core 16.001810 16.002144 16.003226 16.002191 16.002365 16.012578
##      two_cores  8.000865  8.001134  8.001223  8.001196  8.001423  8.001467
##     four_cores  4.000583  4.000688  4.000996  4.000815  4.000864  4.003108
##    eight_cores  2.000543  2.001572  2.006912  2.004795  2.010521  2.020339
##  sixteen_cores  1.000396  1.000758  1.004049  1.001803  1.003307  1.021746
##  neval
##     10
##     10
##     10
##     10
##     10
```

We see that the `say_hello_omp` function increases in speed linearly with the number of cores. Generally, this is not true. The machine used to produce this report only has four cores, and thus specifying `ncores`>4 will not increase performace *in general*. This is a unique example as there is *no* computation required at each step, and thus the threads do not have to wait for resources to become available before they are executed.

We can check that this claim is not true in general: we write a function which computes matrix products.

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

using namespace Rcpp;

// [[Rcpp::plugins(openmp)]]

// [[Rcpp::export(mat_mul)]]
NumericMatrix mat_mul(NumericMatrix A, NumericMatrix B, int ncores)
{
  int nrow_a = A.nrow();
  int ncol_b = B.ncol();
  NumericMatrix out(nrow_a, ncol_b);

  #if defined(_OPENMP)
   #pragma omp parallel num_threads(ncores)
   #pragma omp for
  #endif
  for(int i=0; i<nrow_a; i++)
  {
    for(int j=0; j<ncol_b; j++)
    {
      out(i, j) = sum(A.row(i) * B.column(j));
    }
  }

  return out;
}
')
```

```
A <- B <- matrix(rnorm(300 * 300, 4, 1), 300, 300)
microbenchmark(one_core = mat_mul(A, B, 1),
  two_cores = mat_mul(A, B, 2),
  four_cores = mat_mul(A, B, 4),
  eight_cores = mat_mul(A, B, 8),
  sixteen_cores = mat_mul(A, B, 16))
```

```
## Unit: milliseconds
##           expr       min        lq     mean   median       uq      max neval
##       one_core 31.258242 34.642952 39.05817 37.46289 41.53266 79.11157   100
##      two_cores 17.491381 18.907508 22.31084 20.36666 23.33258 62.78430   100
##     four_cores  9.237565 11.076467 14.45241 14.83699 16.41447 30.21558   100
##    eight_cores  8.471450  9.688336 15.03848 11.19382 14.61013 58.39807   100
##  sixteen_cores  8.395031 10.115915 12.08050 11.20853 12.33696 38.33617   100
```

In this case we see that optimal performance is achieved when using four cores. This is because there is overhead associated with using a larger number of threads, and this overhead increases whilst the number of threads increases, but the as the machine has four physical cores the task does not become easier when more than four threads are used.

When attempting parallelism in Rcpp we need to be careful as the Rcpp API is not necessarily thread-safe. In the above `mat_mul` function we used `out(i, j) = sum(A.row(i) * B.column(j))` and parallelized over the outer loop (over i). This operation *is* thread-safe as each thread is updating an element of the matrix `out` and there is no possibility that multiple threads will try to update an object at the same time.

Below we implement a function which simply sums the elements of a vector, and use the function to demonstrate the need for thread-safety.

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

using namespace Rcpp;

// [[Rcpp::plugins(openmp)]]

// [[Rcpp::export(sum_vector)]]
double sum_vector(NumericVector A, int ncores)
{
  int n = A.length();
  double out;

  #if defined(_OPENMP)
   #pragma omp parallel num_threads(ncores)
   #pragma omp for
  #endif
  for(int i=0; i<n; i++)
  {
    out += A(i);
  }

  return out;
}
')
```

Note that in the above code we have used `out += A(i)`, which is *not* thread-safe. It is highly likely that multiple threads will try to update `out` at the same time. To test the function, we generate a vector of `1`s of length 250. Summing the elements of this vector should return 250.

```
vect <- rep(1, 250)

cat("Sum using one core:", sum_vector(vect, 1), "\n")
```

```
## Sum using one core: 250
```

```
cat("Sum using two cores:", sum_vector(vect, 2), "\n")
```

```
## Sum using two cores: 250
```

```
cat("Sum using three cores:", sum_vector(vect, 3), "\n")
```

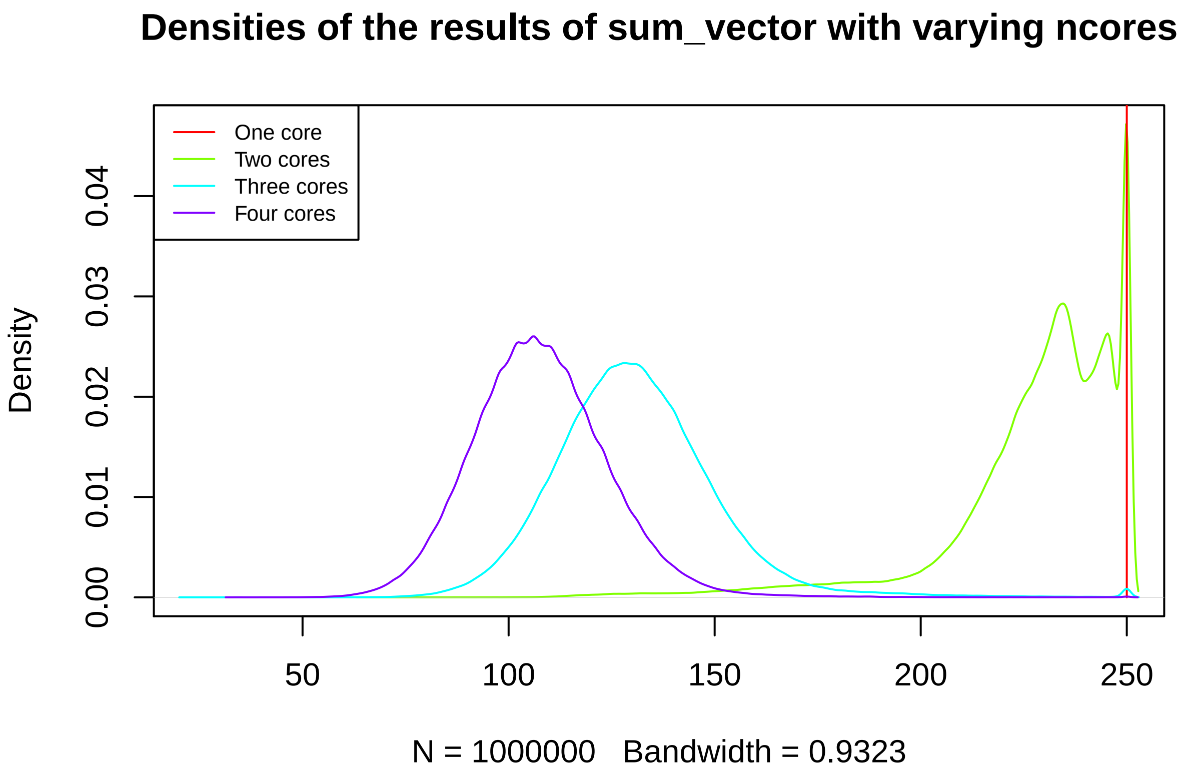
```
## Sum using three cores: 133
```

```
cat("Sum using four cores:", sum_vector(vect, 4), "\n")
```

```
## Sum using four cores: 108
```

As seen above, even simple operations such as summing elements of a vector may fail if the operation is not thread-safe. Below we plot the densities of the outputs of `sum_vector` over a million runs, for a varying number of cores. Note that the output of `sum_vector` is always correct when only using one core.

```
# run test
ntests <- 1e6
test <- matrix(NA, nrow=ntests, ncol=4)
for(i in 1:ntests){
  for(j in 1:4){
    test[i, j] <- sum_vector(vect, j)
  }
}

# plot densities
plot(density(test[,2]), xlim = c(min(test), max(test)), col = rainbow(4)[2],
  main = "Densities of the results of sum_vector with varying ncores")
lines(x=test[c(1,ntests),1], y=seq(0, 1, length.out=2), col = rainbow(4)[1])
lines(density(test[,3]), col = rainbow(4)[3])
lines(density(test[,4]), col = rainbow(4)[4])
legend("topleft", legend = c("One core", "Two cores", "Three cores", "Four cores"),
  cex = 0.7, lty = rep(1,4), col = rainbow(4))
```



Parallel random number generation using OpenMP

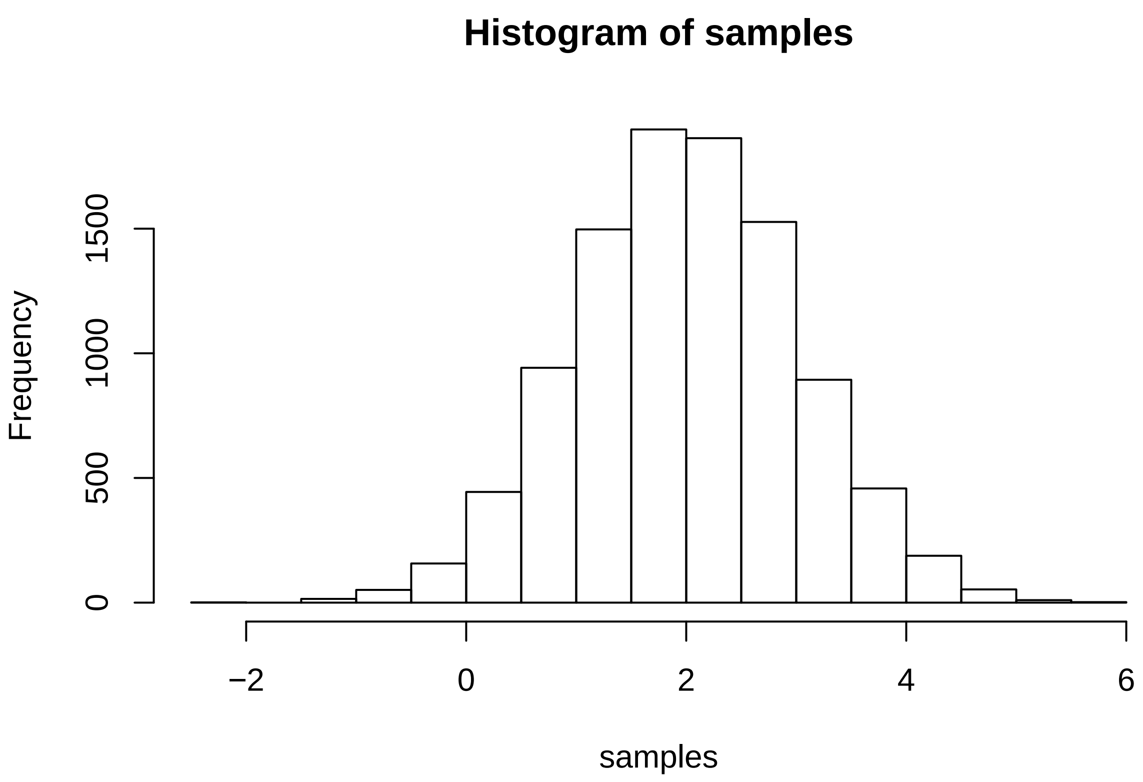
R's RNG is not thread-safe and so we will demonstrate how we can generate random variables in a thread-safe way by using the `sitmo` package. Below is a simple function which produces Gaussian random variables using the `sitmo` package. We use the `sitmo` package to generate uniform random variables, apply the Box-Muller transform to generate iid samples from a standard Gaussian distribution, and apply a transformation to produce Gaussian random variables with the required mean and variance.

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

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

// [[Rcpp::export(rnorm_sitmo)]]
Rcpp::NumericVector rnorm_sitmo(int n, double mu, double sd,
                                  double seed) {
  Rcpp::NumericMatrix samples(ceil(n/2.0), 2);
  Rcpp::NumericVector out(n);

  uint32_t coreseed = static_cast<uint32_t>(seed);
  sitmo::prng eng(coreseed);

  double mx = sitmo::prng::max();

  for(unsigned int i=0; i<ceil(n/2.0); i++)
  {
    double u1 = eng() / mx;
    double u2 = eng() / mx;
    double z1 = sqrt(-2*log(u1))*cos(2*M_PI*u2);
    double z2 = sqrt(-2*log(u1))*sin(2*M_PI*u2);
    samples(i, 0) = z1;
    samples(i, 1) = z2;
  }

  for(int i=0; i<n; i++)
  {
    if(i < ceil(n/2.0)) out(i) = samples(i, 0);
    if(i >= ceil(n/2.0)) out(i) = samples(i - ceil(n/2.0), 1);
  }

  out = sd*(out + mu);

  return out;
}
')
```

In the above code we specify that the function depends on the `sitmo` package via `// [[Rcpp::depends(sitmo)]]`. We convert the `int seed` input to a `uint32_t` object, and set up a RNG with the seed using `sitmo::prng eng(coreseed);`. The RNG provided by `sitmo` will generate a uniform random variable between 0 and `sitmo::prng::max()`, and so when we generate a random variable via `eng()`, we divide it by `sitmo::prng::max()`. We apply the Box-Muller transform to generate pairs of Gaussian random variables using the uniform random variables, and output the samples.

We check that the function works as expected.

```
samples <- rnorm_sitmo(10000, 2, 1, 1)
hist(samples)
```



We now use OpenMP to parallelize the above code. This is possible because the RNG provided by the `sitmo` package is thread-safe.

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

#ifdef _OPENMP
  #include <omp.h>
#endif

// [[Rcpp::depends(sitmo)]]
// [[Rcpp::plugins(openmp)]]

// [[Rcpp::export(rnorm_sitmo_omp)]]
Rcpp::NumericVector rnorm_sitmo_omp(int n, double mu, double sd,
                                  Rcpp::NumericVector seeds) {

  Rcpp::NumericVector out(n);
  Rcpp::NumericMatrix samples(ceil(n/2.0), 2);
  int ncores = seeds.size();

  #ifdef _OPENMP
    #pragma omp parallel num_threads(ncores)
  {
  #endif

  uint32_t coreseed = static_cast<uint32_t>(seeds[0]);

  #ifdef _OPENMP
```

```
    coreseed = static_cast<uint32_t>(seeds[omp_get_thread_num()]);
  #endif

  sitmo::prng eng(coreseed);

  double mx = sitmo::prng::max();
  int loopmax = ceil(n/2.0);

  #ifdef _OPENMP
    #pragma omp for
  #endif
  for(unsigned int i=0; i<loopmax; i++)
  {
    double u1 = eng() / mx;
    double u2 = eng() / mx;
    double z1 = sqrt(-2*log(u1))*cos(2*M_PI*u2);
    double z2 = sqrt(-2*log(u1))*sin(2*M_PI*u2);
    samples(i, 0) = z1;
    samples(i, 1) = z2;
  }

  #ifdef _OPENMP
    #pragma omp for
  #endif
  for(unsigned int i=0; i<n; i++)
  {
    if(i < loopmax) out(i) = samples(i, 0);
    if(i >= loopmax) out(i) = samples(i - loopmax, 1);
  }

  #ifdef _OPENMP
  }
  #endif

  out = sd*(out + mu);

  return out;
}
')
```

To parallelize the function with OpenMP, we have used the following:

```
#ifdef _OPENMP
  #pragma omp parallel num_threads(ncores)
{
#endif
```

This introduces a new parallel scope in which the variables declared such as the `coreseed` are private to each thread.

```
#ifdef _OPENMP
  coreseed = static_cast<uint32_t>(seeds[omp_get_thread_num()]);
#endif
```

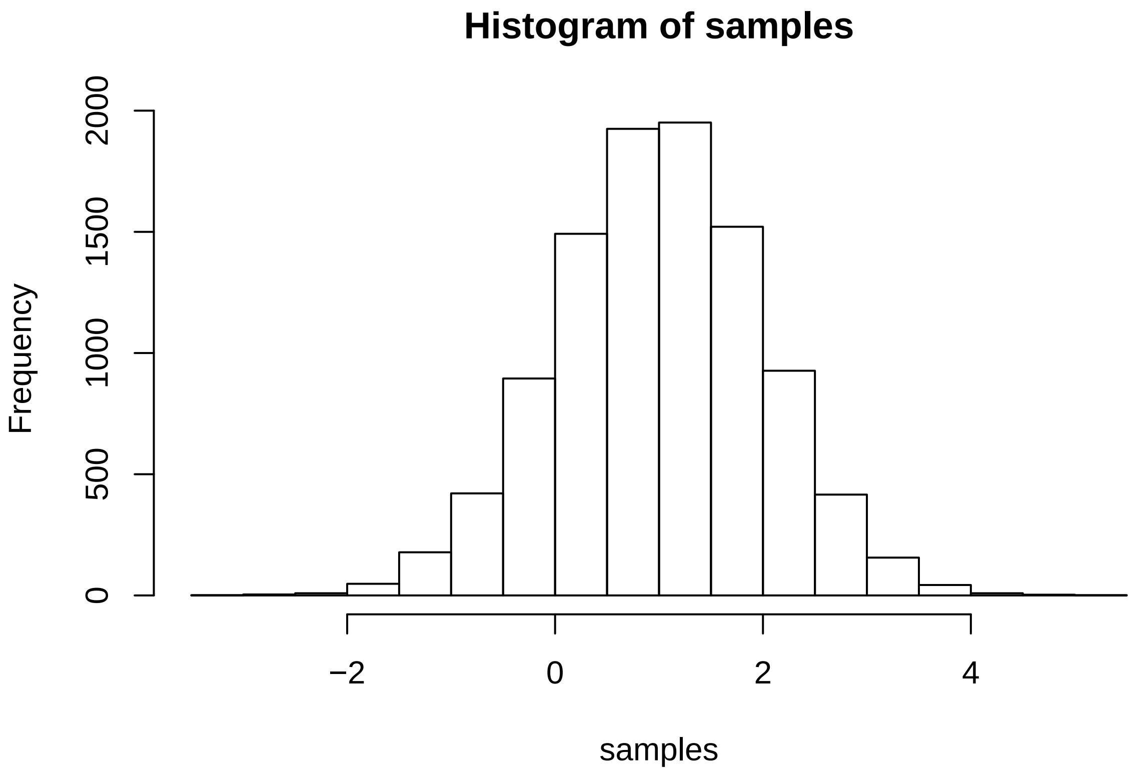
This ensures that the RNG associated with each thread has a different seed, and thus the threads produce different samples.

Within this scope, we indicate the beginning of a parallel for loop as usual with

```
#ifdef _OPENMP
  #pragma omp for
#endif
```

We can check that the function `rnorm_sitmo_omp` *does* produce samples from a Gaussian distribution.

```
samples <- rnorm_sitmo_omp(10000, 1, 1, 1:4)
hist(samples)
```



We can compare the speed to `R`'s built-in `rnorm` function.

```
n <- 1e3
microbenchmark(R = rnorm(n, 0, 1),
  C_one_core = rnorm_sitmo_omp(n, 0, 1, 1),
  C_two_cores = rnorm_sitmo_omp(n, 0, 1, 1:2),
  C_four_cores = rnorm_sitmo_omp(n, 0, 1, 1:4),
  C_sixteen_cores = rnorm_sitmo_omp(n, 0, 1, 1:16), unit="relative")
```

```
## Unit: relative
##             expr       min        lq       mean   median        uq        max
##                R 1.9686970 1.4321593  0.6410488 1.065981  0.975494 0.05920589
##       C_one_core 1.8100231 1.5281735  0.6514271 1.149005  0.955266 0.04935118
##      C_two_cores 1.0000000 1.0000000  1.0000000 1.000000  1.000000 1.00000000
##     C_four_cores 0.6494988 0.9051581  4.0795755 1.065685  1.293130 1.97157692
##  C_sixteen_cores 3.3133385 4.0261857 11.4146825 5.408006 19.109095 3.20705280
##  neval
##    100
##    100
##    100
##    100
##    100
```

```
n <- 1e6
microbenchmark(R = rnorm(n, 0, 1),
  C_one_core = rnorm_sitmo_omp(n, 0, 1, 1),
  C_two_cores = rnorm_sitmo_omp(n, 0, 1, 1:2),
  C_four_cores = rnorm_sitmo_omp(n, 0, 1, 1:4),
  C_sixteen_cores = rnorm_sitmo_omp(n, 0, 1, 1:16), unit="relative")
```

```
## Unit: relative
##             expr      min       lq     mean   median       uq       max neval
##                R 3.743703 3.504881 3.071343 3.350136 2.868309 0.9300673   100
##       C_one_core 3.844929 3.585842 3.148578 3.403521 2.983419 0.9354266   100
##      C_two_cores 1.994045 1.921468 1.790665 1.868984 1.706260 1.0592910   100
##     C_four_cores 1.160140 1.209613 1.241524 1.333155 1.269688 0.9767654   100
##  C_sixteen_cores 1.000000 1.000000 1.000000 1.000000 1.000000 1.0000000   100
```

We see that when generating a small number of samples, the overhead of assigning threads outweighs the computational cost and R's built-in `rnorm` function is much faster. However, for a large number of samples the `rnorm_sitmo_omp` function is faster.

Parallel Rcpp

In this section we demonstrate the use of RcppParallel for parallelization. As seen previously, R's C API can lead to misleading results if we do not ensure that we manipulate objects such as vectors in a thread-safe way — we saw that simply summing the elements of a vector can often lead to errors. `RcppParallel` provides data structures which are accessed in a thread-safe way, and so we can parallelize our code without difficulty. The package also allows us to parallelize code at a higher level, without specifying scopes which are unique for different threads.

`RcppParallel` provides thread-safe accessors which are simply wrappers around underlying R vectors and matrices. These are:

- `RVector<T>` — Wraps R vectors.
- `RMatrix<T>` — Wraps R matrices.

Their use is very intuitive, and provide an easy approach to ensure that data is retrieved and stored in a thread-safe way.

`RcppParallel` also provides tools for parallelism. To demonstrate the use of parallel for loops we will generate a Gram matrix using a Gaussian kernel. The Gram matrix K has elements $K_{i,j} = k(x_i, x_j)$, $i, j = 1, \ldots, n$, where k is a kernel function. The Gaussian kernel function can be defined as

$$k(x,y) = \exp\left(-\frac{1}{2\gamma^2}\|x-y\|^2\right).$$

The norm $\|\cdot\|$ used in the kernel is the L2 norm and so x and y can be in $\mathbb{R}^d$ for arbitrary d, but we will only consider univariate inputs. A simple implementation of this in R can be found below.

```
kernel_gaussian <- function(x, y, gamma) return(exp(-0.5 / gamma^2 * (x - y)^2))
make_Gram <- function(x, gamma){
  kernel.fn <- function(x, y) kernel_gaussian(x, y, gamma)
  return(outer(x, x, kernel.fn))
}
```

We can show that the function `make_Gram` works as expected.

```
x <- seq(-10, 10, length.out = 4)
make_Gram(x, 4)
```

```
##              [,1]       [,2]       [,3]         [,4]
## [1,] 1.000000e+00 0.24935221 0.00386592 3.726653e-06
## [2,] 2.493522e-01 1.00000000 0.24935221 3.865920e-03
## [3,] 3.865920e-03 0.24935221 1.00000000 2.493522e-01
## [4,] 3.726653e-06 0.00386592 0.24935221 1.000000e+00
```

A very simple Rcpp implementation (without parallelization) can be found below. To produce the Gram matrix we simply use a nested for loop to compute evaluations of the kernel function.

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

double kernel_gaussian(double x, double y, double gamma)
{
  return exp(-0.5 / pow(gamma, 2) * pow(x - y, 2));
}

// [[Rcpp::export]]
Rcpp::NumericMatrix make_gram_rcpp(Rcpp::NumericVector x, double gamma) {

  int n = x.length();
  Rcpp::NumericMatrix out(n, n);

  for(int i=0; i<n; i++)
  {
    for(int j=0; j<n; j++)
    {
      out(i, j) = kernel_gaussian(x(i), x(j), gamma);
    }
  }

  return out;
}
')
```

We can use `RcppParallel` to parallelize the for loop as follows.

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

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

double kernel_gaussian(double x, double y, double gamma)
{
  return exp(-0.5 / pow(gamma, 2) * pow(x - y, 2));
}

struct kernel_eval : public Worker
{
   const RVector<double> in;
   const double gamma;
   RMatrix<double> out;

   kernel_eval(const Rcpp::NumericVector in_, double gamma, Rcpp::NumericMatrix out_)
      : in(in_), gamma(gamma), out(out_) {}

   void operator()(std::size_t begin, std::size_t end) {
     for(std::size_t i = begin; i < end; i++){
       for(std::size_t j = 0; j < in.length(); j++){
         out(i, j) = kernel_gaussian(in[i], in[j], gamma);
       }
     }
   }
};

// [[Rcpp::export]]
Rcpp::NumericMatrix make_gram_par(Rcpp::NumericVector x, double gamma) {

  int n = x.length();
  Rcpp::NumericMatrix out(n, n);

  kernel_eval obj(x, gamma, out);

  parallelFor(0, x.length(), obj);

  return out;
}
')
```

The code above is parallelized by the `parallelFor` function provided by the `RcppParallel` package. The first two arguments of the function specify the beginning and end of the for loop, and the third input is an object with the `RcppParallel::Worker` type which we explain below. The function also has an optional argument `grainSize` which allows us to specify how many iterations each thread should do. To create the object of type `RcppParallel::Worker` we have specified the data structure `kernel_eval` which inherits the `Worker` type. The structure has three elements: the input, the output, and the kernel length-scale parameter γ. The elements are intialized by the constructor

```
 kernel_eval(const Rcpp::NumericVector in_, double gamma, Rcpp::NumericMatrix out_)
      : in(in_), gamma(gamma), out(out_) {}
```

which we call in the `make_gram_par` function as follows

```
int n = x.length();
Rcpp::NumericMatrix out(n, n);
kernel_eval obj(x, gamma, out);
```

This initializes the memory for the output matrix and provides the input data x, length-scale parameter γ, and the output vector `out` to the constructor. The constructor wraps the input and output with thread-safe accessors via `RVector<double>` and `RMatrix<double>`. The structure also contains the operator (a simple nested for loop in this case) which will be parallelized:

```
void operator()(std::size_t begin, std::size_t end) {
 for(std::size_t i = begin; i < end; i++){
   for(std::size_t j = 0; j < in.length(); j++){
     out(i, j) = kernel_gaussian(in[i], in[j], gamma);
   }
 }
}
```

When running the function `make_Gram_par`, a worker (a thread) will be assigned a `begin` and `end` with which it will run the `operator()` with. If we use the default `grainSize` of 1, then `end` = `begin` + 1, and each worker will run the outer for loop for 1 iteration which corresponds to each worker computing a *row* of the Gram matrix.

```r
x <- seq(-10, 10, length.out=300)
microbenchmark(make_Gram(x, 4),
  make_gram_rcpp(x, 4),
  make_gram_par(x, 4),
  unit = "relative")
```

```
## Unit: relative
##                  expr      min       lq     mean   median       uq      max
##       make_Gram(x, 4) 4.054229 3.469162 3.233931 3.484863 3.625258 3.153007
##  make_gram_rcpp(x, 4) 4.019898 3.193696 2.821271 3.145319 3.275899 1.828661
##   make_gram_par(x, 4) 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
##  neval
##    100
##    100
##    100
```