portfolio_9

2023-05-09

Parrallel Rcpp

In this portfolio we will look at how to carry out parrallel computation in R using Rcpp. We will look at an Irish smart meter data set and do some analysis on electricity demand using the help of parralelization in Rcpp. First lets load in the data:

```
library(devtools)
```

```
## Loading required package: usethis
```

```
install_github("mfasiolo/electBook")
```

```
## Skipping install of 'electBook' from a github remote, the SHA1 (38d7020f) has not changed since last
##   Use `force = TRUE` to force installation
```

```
library(electBook)
```

```
## Registered S3 method overwritten by 'quantmod':
##   method            from
##   as.zoo.data.frame zoo
```

```
data(Irish)
```

Let's now concatenate all of the electricity demand from all households into a single vector and print out some of the first values of the vector:

```
y <- do.call("c", Irish$indCons)
y <- y - mean(y)
head(y)
```

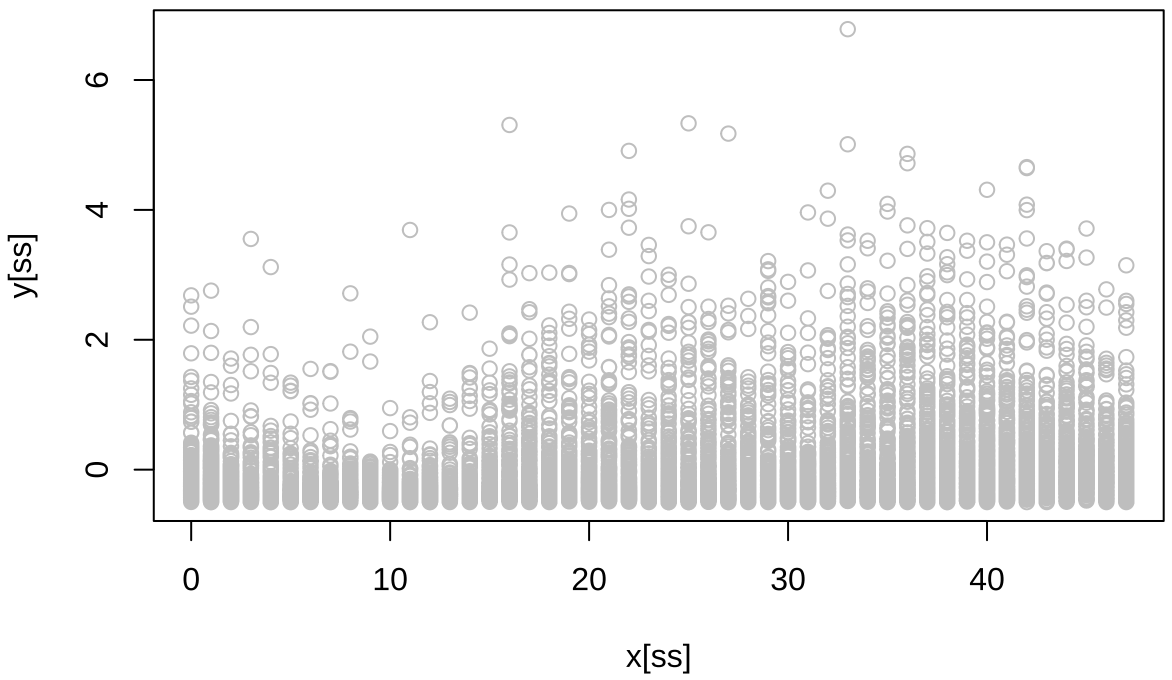
```
##     I10021     I10022     I10023     I10024     I10025     I10026
## -0.4773827 -0.3663827 -0.4053827 -0.4763827 -0.3663827 -0.4093827
```

We would like to model y, the electricity demand, according to the time of day which is given by the vector x:

```
ncust <- ncol(Irish$indCons)
x <- rep(Irish$extra$tod, ncust)
```

Let's now plot a subsample of the data:

```
n <- length(x)
ss <- sample(1:n, 1e4)
plot(x[ss], y[ss], col = "grey")
```



Here y is the electricity demand and x is the time of day, 0 corresponds to midnight and 47 23:30, ie. x represents the time of day as the number of 30 minute intervals that have so far passed. We would like to fit a simple univariate regression model $\mathbb{E}(y|x) = \beta_1 x$. Let's first create a function in regular R that would do this:

```r
reg1D <- function(y, x){

  b <- t(x) %*% y / (t(x) %*% x)

  return(b)

}
```

Let's now create a version in RcppParallel, so we can split computation across multiple cores:

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

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

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

```r
Rcpp::sourceCpp(code = '
// [[Rcpp::depends(RcppParallel)]]
#include <Rcpp.h>
#include <RcppParallel.h>
#include <iostream>

using namespace Rcpp;
using namespace RcppParallel;

// Define a struct to hold the input and output data
struct RegressionData {
  const RVector<double> x;
  const RVector<double> y;
  double xx;
  double xy;

  // Constructor to initialize the input data
  RegressionData(const NumericVector x_, const NumericVector y_)
    : x(x_), y(y_), xx(0), xy(0) {}
};

// Define the worker function to calculate the sums
struct RegressionWorker : public Worker {
  RegressionData* data;

  // Constructor to initialize the worker with the input data
  RegressionWorker(RegressionData* data_) : data(data_) {}

  // Function to perform the calculation
  void operator()(std::size_t begin, std::size_t end) {
    double partial_xx = 0.0;
    double partial_xy = 0.0;
    for (std::size_t i = begin; i < end; i++) {
      partial_xx += (data->x[i] * data->x[i]);
      partial_xy += (data->x[i] * data->y[i]);
    }
    #pragma omp critical
    {
      data->xx += partial_xx;
      data->xy += partial_xy;
    }
  }
};

// [[Rcpp::export]]
NumericVector reg1D_parallel(NumericVector y, NumericVector x) {

  // Create RegressionData object with input data
  RegressionData data(x, y);

  // Create RegressionWorker object with pointer to RegressionData
  RegressionWorker worker(&data);

  // Call parallelReduce to perform the calculation in parallel
  parallelFor(0, x.length(), worker);

  // Perform: xy / xx
  NumericVector result(1);
  #pragma omp critical
  {
    result[0] = data.xy / data.xx;
  }
  return result;
}
```

```
')
```

Let's first compare the fits of these two models and see if they reach the same conclusion:

```
lm(y ~ -1 + x)$coeff
```

```
##           x
## 0.003250122
```

```
reg1D(y, x)
```

```
##             [,1]
## [1,] 0.003250122
```

```
reg1D_parallel(y, x)
```

```
## [1] 0.003249945
```

The results are very similar with some minor discrepancies for the last 4 digits. Let's now compare the speed of these two functions in fitting a model, we will also set the number of threads to be used to 8, our parallel function will use the maximum number of threads available so will use 8 threads:

```
setThreadOptions(numThreads = 8)

# Fit using both functions and print times
system.time( reg1D(y, x) )
```

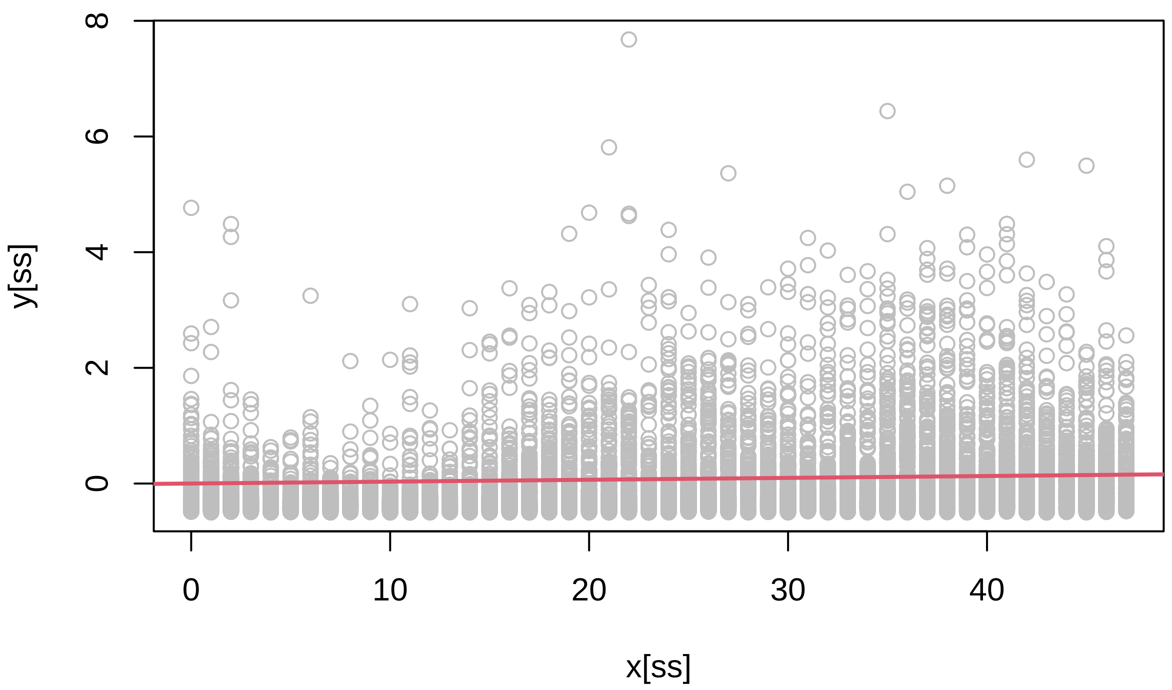
```
##    user  system elapsed
##   0.329   0.120   0.453
```

```
system.time( reg1D_parallel(y, x) )
```

```
##    user  system elapsed
##   0.175   0.000   0.023
```

We can see that the time for our parallel version of the function to run was much less than the time needed for the non-parallel version. Let's now take a look at the model that is fitted by reg1D_parallel:

```
ss <- sort( sample(1:n, 1e4) )
plot(x[ss], y[ss], col = "grey")
abline(a = 0, b = reg1D_parallel(y, x), col = 2, lwd = 2)
```



The fit is t errible, this is pretty obvious simply by taking a look at the scatter plot. Let's now instead try and fit a non-linear model that may more accurately be able to model the electricity demand. Let's now try fitting a polynomial regression of the following form, $\mathbb{E}(y|x) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$. First let's write our non-parallel version:

```r
regMD <- function(y, X){

  XtX <- t(X) %*% X

  C <- chol(XtX)

  b <- backsolve(C, forwardsolve(t(C), t(X) %*% y) )

  return(b)

}
```

And now let's write our parallel version, we will write functions for computing $X^T X$ and finding the Choleksy decomposition:

```r
Rcpp::sourceCpp(code = '
// [[Rcpp::depends(RcppParallel)]]
#include <Rcpp.h>
#include <RcppParallel.h>
#include <cmath>
#include <iostream>
using namespace Rcpp;
using namespace RcppParallel;

// Define a custom worker class for parallel computation of Cholesky decmposition
class CholeskyWorker : public Worker {
private:
  const RMatrix<double> A;
  RMatrix<double> R;
```

```
public:
  CholeskyWorker(const NumericMatrix A, NumericMatrix R)
    : A(A), R(R) {}

  // Function to perform the parallel computation
  void operator()(std::size_t begin, std::size_t end) {
    std::size_t n = A.nrow();

    for (std::size_t i = begin; i < end; ++i) {
      for (std::size_t j = i; j < n; ++j) {
        double sumProduct = 0.0;
        for (std::size_t k = 0; k < i; ++k) {
          sumProduct += R(k, i) * R(k, j);
        }
        if (i == j) {
          R(i, i) = std::sqrt(A(i, i) - sumProduct);
        } else {
          R(i, j) = (A(i, j) - sumProduct) / R(i, i);
        }
      }
    }
  }
};

// Function to perform Cholesky decomposition in parallel
// [[Rcpp::export]]
NumericMatrix cholParallel(const NumericMatrix A) {
  std::size_t n = A.nrow();
  NumericMatrix R(n, n);

  // Create an instance of the CholeskyWorker
  CholeskyWorker worker(A, R);

  // Perform parallel computation using RcppParallel
  parallelFor(0, n, worker);

  // Set the lower triangular elements to zero
  for (std::size_t i = 0; i < n; ++i) {
    for (std::size_t j = 0; j < i; ++j) {
      R(i, j) = 0.0;
    }
  }

  return R;
}

// Define a custom worker class for parallel computation
class XtXWorker : public Worker {
private:
  const RMatrix<double> X;
  RMatrix<double> result;

public:
  XtXWorker(const NumericMatrix X, NumericMatrix result)
    : X(X), result(result) {}

  // Function to perform the parallel computation
  void operator()(std::size_t begin, std::size_t end) {
    std::size_t n = X.ncol();

    for (std::size_t i = begin; i < end; ++i) {
      for (std::size_t j = i; j < n; ++j) {
        double dotProduct = 0.0;
        for (std::size_t k = 0; k < X.nrow(); ++k) {
          dotProduct += X(k, i) * X(k, j);
        }
        result(i, j) = dotProduct;
        result(j, i) = dotProduct;  // Set the symmetric element as well
      }
    }
  }
};

// Function to perform matrix times itself in parallel
// [[Rcpp::export]]
NumericMatrix XtX(const NumericMatrix X) {
  std::size_t n = X.ncol();
  NumericMatrix result(n, n);

  // Create an instance of the XtXWorker
  XtXWorker worker(X, result);

  // Perform parallel computation using RcppParallel
  parallelFor(0, n, worker);

  return result;
}

')
```

Let's now write the parallel version of our function using the functions we created using RcppParallel above, note that we perform the frontsolve and backsolve like in the non-parallel version as these functions are sequential and therfore cant be paralellized:

```
regMD_parallel <- function(y, X){
  C <- cholParallel(XtX(X))
  b <- backsolve(C, forwardsolve(t(C), t(X) %*% y) )
  return(b)
}
```

Let's now check that the values we get are the same:

```
X <- cbind(1, x, x^2, x^3)
b_lm <- lm(y ~ -1 + X)$coeff; b_lm
```

```
##             X            Xx             X             X 
## -1.238304e-01 -3.216597e-02  2.354865e-03 -3.292171e-05
```

```
b <- regMD(y, X) ; b
```

```
##               [,1]
## [1,] -1.238304e-01
## [2,] -3.216597e-02
## [3,]  2.354865e-03
## [4,] -3.292172e-05
```

```
b_p <- regMD_parallel(y, X) ; b_p
```

```
##               [,1]
## [1,] -1.238304e-01
## [2,] -3.216597e-02
## [3,]  2.354865e-03
## [4,] -3.292172e-05
```

They all look the same! great! Let's now compare their compute times:

```
system.time( regMD(y, X) )
```

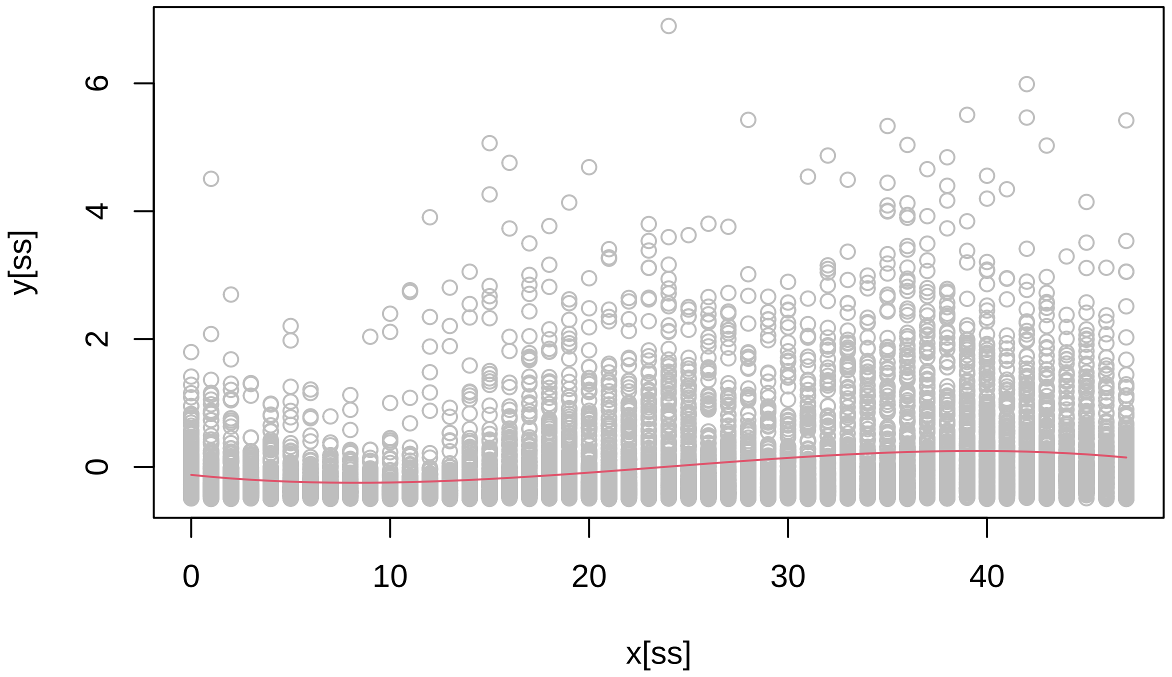
```
##    user  system elapsed
##   1.189   0.508   1.697
```

```
system.time( regMD_parallel(y, X) )
```

```
##    user  system elapsed
##   0.987   0.203   0.827
```

The compute times for our parallel version are quite a bit faster, let's now see what the fit looks like:

```
ss <- sort( sample(1:n, 1e4) )
Xu <- cbind(1, 0:47, (0:47)^2, (0:47)^3)
plot(x[ss], y[ss], col = "grey")
lines(0:47, Xu %*% b_p, col = 2)
```



This fit looks much better than before, but can we do even better? let's now try and fit a basic one dimensional local polynomial regression (using a Gaussian kernel), let's first write our non-parallel version:

```r
regMD_local <- function(y, X, x0, x, h){

  w <- dnorm(x0, x, h)
  w <- w / sum(x) * length(y)

  wY <- y * sqrt(w)
  wX <- X * sqrt(w)
  wXtwX <- t(wX) %*% wX

  C <- chol(wXtwX)

  b <- backsolve(C, forwardsolve(t(C), t(wX) %*% wY) )

  return(b)

}
```

We already have a function for performing the cholesky decomposition in parallel,

```r
Rcpp::sourceCpp(code = '
// [[Rcpp::depends(RcppParallel)]]
#include <Rcpp.h>
#include <RcppParallel.h>
#include <cmath>
#include <iostream>

using namespace Rcpp;
using namespace RcppParallel;

// [[Rcpp::export]]
double cpp_dnorm(double x, double mu, double sigma) {
    double coefficient = 1 / (sqrt(2 * M_PI) * sigma);
    double exponent = -((x - mu) * (x - mu)) / (2 * sigma * sigma);
    return coefficient * exp(exponent);
}

class VectorSumWorker : public Worker
{
private:
  const RVector<double> input;
  double sum;

public:
  VectorSumWorker(const NumericVector input) : input(input), sum(0.0) {}

  VectorSumWorker(VectorSumWorker& other, Split) : input(other.input), sum(0.0) {}

  void operator()(std::size_t begin, std::size_t end)
  {
    double localSum = 0.0;
    for (std::size_t i = begin; i < end; ++i)
    {
      localSum += input[i];
    }
    sum += localSum;
  }

  void join(const VectorSumWorker& rhs)
  {
    sum += rhs.sum;
  }

  double getSum() const
  {
    return sum;
  }
};

// [[Rcpp::export]]
double VectorSum(NumericVector input)
{
  VectorSumWorker worker(input);
  parallelReduce(0, input.size(), worker);

  return worker.getSum();
}

class WeightsWorker : public Worker
{
private:
  const double x0;
  const RVector<double> x;
  const double h;
  const double sx;
  const double ylen;
  NumericVector w;

public:
  WeightsWorker(const double x0, const NumericVector x, const double h, const double sx, const double yl
    : x0(x0), x(x), h(h), w(x.size()), sx(sx), ylen(ylen) {}

  void operator()(std::size_t begin, std::size_t end)
  {
    for (std::size_t i = begin; i < end; ++i)
    {
      double weight = cpp_dnorm(x0, x[i], h);
      w[i] = weight/sx * ylen;
    }
  }

  NumericVector getWeights() {
    return w;
  }
};

// [[Rcpp::export]]
NumericVector get_weights(double x0, NumericVector x, double h, NumericVector y)
```

```
{
  double sx  = VectorSum(x);
  double ylen = y.size();
  double c = sx*ylen;
  WeightsWorker weights(x0, x, h, sx, ylen);
  parallelFor(0, x.size(), weights);

  return weights.getWeights();
}

struct MulVecWorker : public Worker {
  const RVector<double> x;
  const RVector<double> y;
  RVector<double> output;

  MulVecWorker(const NumericVector& x, const NumericVector& y, NumericVector& result)
    : x(x), y(y), output(result) {}

  void operator()(std::size_t begin, std::size_t end) {
    for (std::size_t i = begin; i < end; ++i) {
      output[i] = x[i] * y[i];
    }
  }
};

// [[Rcpp::export]]
NumericVector mulVecParallel(const NumericVector& x, const NumericVector& y) {
  // Create an output vector of the same length as input vectors
  NumericVector output(x.size());

  // Create a worker object
  MulVecWorker worker(x, y, output);

  // Perform the parallel calculation
  parallelFor(0, x.size(), worker);

  return output;
}

struct ElementwiseMatVecMulWorker : public Worker {
  const RMatrix<double> inputX;
  const RVector<double> inputY;
  RMatrix<double> output;

  ElementwiseMatVecMulWorker(const NumericMatrix& X, const NumericVector& y, NumericMatrix& result)
    : inputX(X), inputY(y), output(result) {}

  void operator()(std::size_t begin, std::size_t end) {
    for (std::size_t i = begin; i < end; ++i) {
      for (int j = 0; j < inputX.ncol(); ++j) {
        output(i, j) = inputX(i, j) * inputY[i];
      }
    }
  }
};

// [[Rcpp::export]]
NumericMatrix elementwiseMatVecMulParallel(const NumericMatrix& X, const NumericVector& y) {
  NumericMatrix output(X.nrow(), X.ncol());

  ElementwiseMatVecMulWorker worker(X, y, output);

  parallelFor(0, X.nrow(), worker);

  return output;
}

')
```

We use the paralellized functions above in our parallel version of the function:

```
regMD_local_parallel <- function(y, X, x0, x, h){

  w <- get_weights(x0, x, h, y)

  sqrt_w <- sqrt(w)
  wY <- mulVecParallel(y, sqrt_w)
  wX <- elementwiseMatVecMulParallel(X, sqrt_w)
  wXtwX <- XtX(wX)

  C <- cholParallel(wXtwX)

  b <- backsolve(C, forwardsolve(t(C), t(wX) %*% wY) )

  return(b)

}
```

Let's first check if the answers we get back are the same:

```
b_lm <- lm(y ~ -1 + X, weights = dnorm(35, x, 5))$coeff; b_lm
```

```
##             X            Xx             X             X
##  6.9754729881 -0.6886549222  0.0218422872 -0.0002197244
```

```
b <- regMD_local(y, X, x0 = 35, x = x, h = 5) ; b
```

```
##               [,1]
## [1,]  6.9738678855
## [2,] -0.6885095762
## [3,]  0.0218379887
## [4,] -0.0002196828
```

```
b_p <- regMD_local_parallel(y, X, x0 = 35, x = x, h = 5) ; b_p
```

```
##               [,1]
## [1,]  6.9738678856
## [2,] -0.6885095762
## [3,]  0.0218379887
## [4,] -0.0002196828
```

They are all the same! Let's now see how much quicker our parallel version is!

```
system.time( regMD_local(y, X, x0 = 35, x = x, h = 5) )
```

```
##    user  system elapsed
##   2.399   0.964   3.363
```

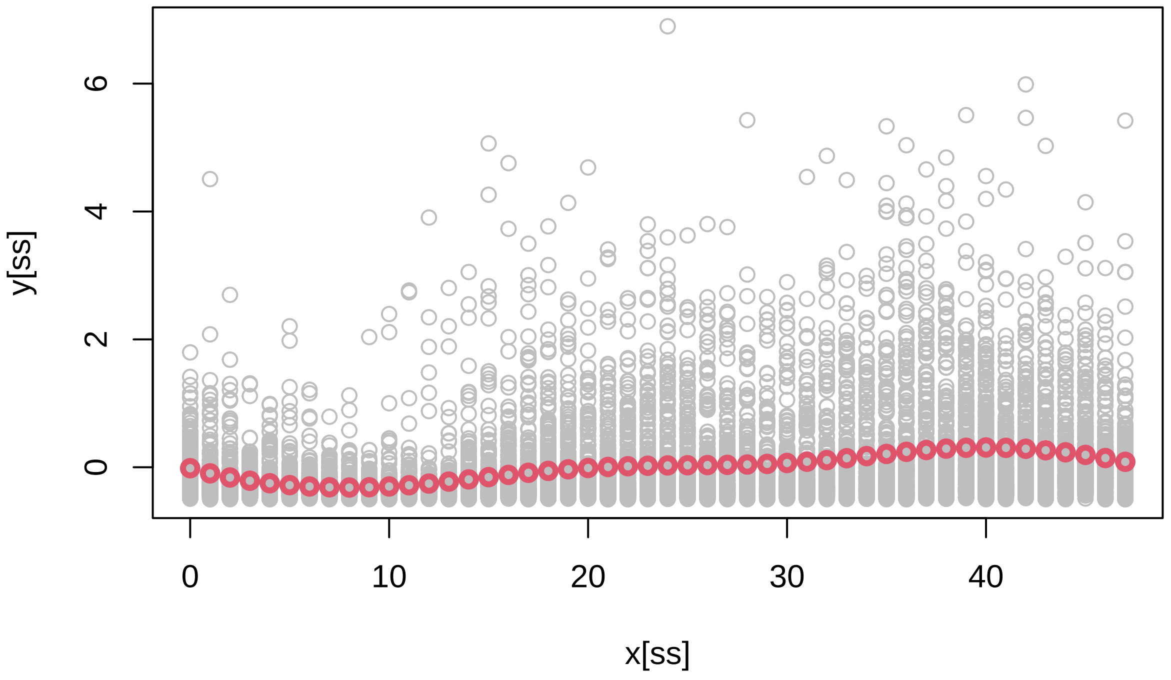
```
system.time( regMD_local_parallel(y, X,  x0 = 35, x = x, h = 5) )
```

```
##    user  system elapsed
##   3.018   0.644   1.678
```

We see the user time has increased slightly due to extra setup, however the total elapsed time is much less than the time taken for our non-parallel version. Let's now see the fit of this model:

```
f <- lapply(0:47, function(.x ) regMD_local_parallel(y, X, x0 = .x, x = x, h = 5))

plot(x[ss], y[ss], col = "grey")
lines(0:47, sapply(1:48, function(ii) Xu[ii, ] %*% f[[ii]]), col = 2, lwd = 3, type = 'b')
```



This fit looks even better than our previous fit!