Parallelization of B-Splines
ECS158 Final Project

Brian Vargas, Max Matsumoto, Michael Banzon, Mark Fan

March 20, 2015

1 Introduction

1.1 Background

Given a set of data, we can use B-Splines as a form of regression. This method essentially creates a variety of piecewise polynomial functions of a given degree to better represent the data. It is often the case that we have too much data to process quickly so we turn to parallel programming to speed up the process.

We have created an R package in which the output of the function is the basis matrix containing the elements found via the Cox-de Boor recursion formula for each given x value.

The Cox-de Boor recursion formula is found as follows[1]:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \le u < u_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)$$

This code could then be applied to regress as follows:

```
n <- 1000
x <- seq(0, 1, length=n)
B <- bs(x) #use default
dgp <- sin(2*pi*x)
y <- dgp + rnorm(n, sd=.1)
model <- lm(y~B-1)
```

[1]http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/B-spline/bspline-basis.html

```
plot(x, y, cex=.25, col="grey")
lines(x, fitted(model), lwd=2)
lines(x, dgp, col="red", lty=2)
```

The code to the functions implemented to generate the B-Spline matrix can be viewed in the appendix for serial R, OpenMP, CUDA, and SNOW.

1.2 Serial R

As a benchmark, we found a simplified version of the B-Spline *bs*() function in serial R[2]. This version heavily relies on the Cox-de Boor recursion formula to generate the B-Spline. This allowed a clear and direct method of calculating the matrix for basis splines. This function was used as the basis of parallelization; modifications were made to exploit each column of the basis matrix to speed up the process while incorporating recursion whenever possible.

1.3 OpenMP

OpenMP allowed the recursive algorithm to be implemented in parallel. As shown in the time trials, this meant significant improvement in computational time. The main set up of the function mirrored the serial function in R with only a few minor changes. The heart of the algorithm still existed in the Cox-de Boor recursive formula. We used an omp parallel for loop through each column oto speed up the formation of the output basis matrix. Inside the parallel loop, an omp critical section is used when storing a returned column to assure no overwriting takes place.

1.4 CUDA

At the core, R uses data type double when dealing with real numbers. CUDA does not support the data type double, and thus, forces a conversion from type double to type float. Due to this conversion, basic memory allocation becomes unpredictable as double is composed of 8-bits as is float composed of 4-bits. This caused many irregularities while copying the arrays for x, knots, and the eventual basis matrix to and from the CUDA device.

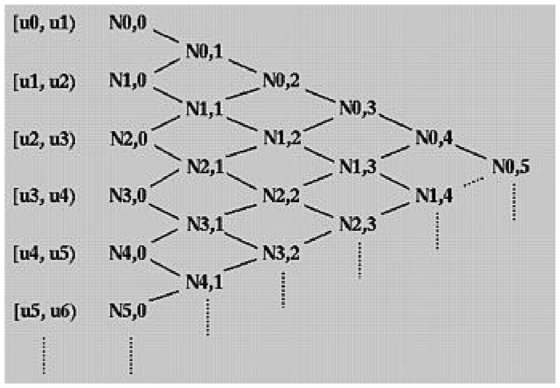
The original approach was to follow a similar algorithm as in the serial R version. However, CUDA does not support recursion, and thus, the algorithm needed modification. Emulating an algorithm optimized for recursion often tends to be difficult and requires much more overhead.

The first approach to an iterative algorithm took use of stacks. The idea was to skip the break down and go directly into the build up. In other words, we were to start from the base case and follow the recursive algorithm in a forward manner exploiting the Cox-de Boors recursion formula in an iterative manner.

[2] `http://cran.r-project.org/web/packages/crs/vignettes/spline_primer.pdf`

For reasons unknown, this approach did not work.

The next approach was to follow a triangular method of computation, filling each index of the return matrix as the algorithm went forward. The Triangular method is described by the following graphic:



Again, this approach was unsuccessful as it a range error.

The final approach was to separate tasks – it would be recursive to the point of the base case. Once entering the base case, when the current degree of the polynomial was equal to zero, knot[i] and knot[i+1] were sent, along with the entire x vector, to the CUDA device. Once on the device, each x value was used in the calculation using parallelization and passed back to the host where it was returned to the basis matrix. Once out of the base case, for any degree greater than zero and less than the given degree, the usual recursive algorithm was used.

As a result, the CUDA algorithm is essentially serial with a small taste of parallel for a minimal speedup. We can still expect improvement from the CUDA algorithm as opposed to the serial version but there is still more optimization that can be done on the CUDA algorithm to make full use of the benefits of a GPU. Therefore, this isn't expected to be a match for the other parallel algorithms but it can at least acheive the job of improving computation time by a bit.

1.5 SNOW

SNOW also exploits the recursive algorithm to find each element in the matrix. SNOW also has the advantage of using the vectorization style that the R programming language has been optimized for, which can give better computational times. The main setup of the function mirrors the serial function in R with each cluster working on a different column of the final output matrix. It gives the worker clusters all the necessary information and then has them find the elements of the particular column they were assigned to. The code gives the user the option to select how many clusters to use and the code is designed to automatically make the clusters for the user.

2 Time Trials

Upon implementing the algorithms, we test the time-elapsed for each algorithm to see which would be best to use in practice. Each time we ran a test, the function was called using equidistant x values between 0 and 1 of length n, B-spline regression polynomial degree 5, intercept included, and boundary knots of 0 and 1 as its parameters. The call to our function was made as follows:

```
bs(x, degree=5,intercept=TRUE, Boundary.knots=c(0,1),
   type="serial",ncls=2)
```

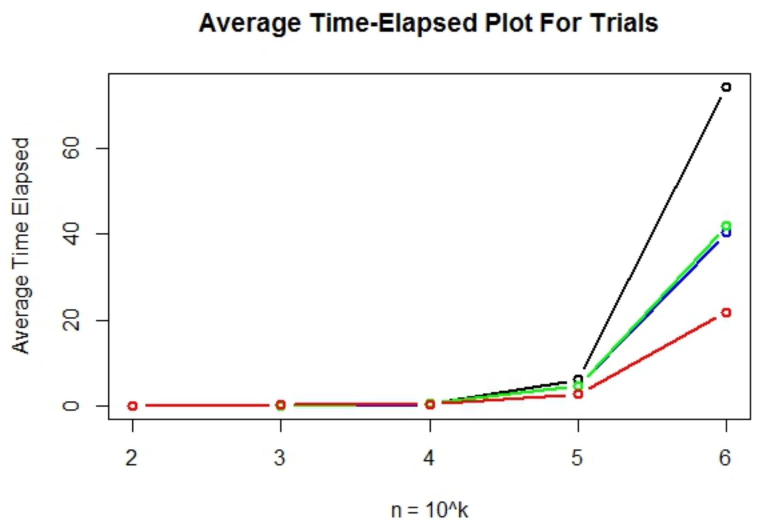
It is important to note that the input values we used for our time trials are irrelevant. We could have used a variation of values or included internal knots but the flops count would still remain the same. Therefore, the trial inputs were kept simple but consistent to make this section easier.

In doing the time trials, for each length n, we call the respective function for either R, OpenMP, CUDA, or SNOW three times and then we took the average of the total time-elapsed. The time is measured in seconds, rounded to the thousandth decimal place.

The following table demonstrates the time trials with varying lengths n:

n =	100	1000	10000	100000	1000000
R	0.015	0.063	0.542	6.179	74.425
OpenMP	0.004	0.037	0.447	4.682	40.562
CUDA	0.048	0.084	0.528	4.647	41.931
SNOW	0.085	0.186	0.358	2.690	21.881

In order to make it easier to view and understand the time trial results, the table can best be summarized by the following plot:



A few interesting observations can be made about the time trials. In the case of OpenMP compared to serial R, when the data set approaches 10000, the difference in time decreases compared to the smaller and the larger n. Surprisingly, starting off much slower than both the serial R function and the OpenMP, the CUDA algorithm is significantly faster than the serial R function and on par with OpenMP as length n increases. Because CUDA is essentially serial, it is possible CUDA's nvcc compiler is much more efficient than the typical GNU g++ compiler used in building OpenMP. As n grows much larger, SNOW takes about half the computational time as OpenMP and CUDA, and a little more that a third of the time the serial R took.

3 Conclusion

As we see in the time trials, SNOW performed the best in each case. When we work with a million x values, it proves to be the very best. OpenMP and CUDA were neck to neck and the serial version was the slowest, as expected. Possibly due to the optimization in R's vectorization style as a programming language, computations were done faster than its C++ counterparts. Unless these C++ interfaces get optimized further, in practice we would want to stick to using R for the parallelization of this problem.

Nonetheless, we definitely achieved the speed up we were looking for: all parallel algorithms did much better than the serial version while maintaining the accuracy. However, we could have done better yet! In our testing, the length n was much greater than the number of columns in the final basis matrix. Since each element in the matrix can technically be found independently of the other elements, we could have done the parallelization over the rows instead of the columns - or whichever is the largest of the two in practice. This would make it so there is less to do for each parallelization but there are more tasks to accomplish as opposed to its current style which would make it so some of the workers remain inactive since there are such few tasks.

A Source Code

The following source code in R and C++ extensions make up our package *parallelBS*. It can be loaded using the following R command:

```
library(parallelBS)
```

A.1 Driver

This is how the function in our package is run. Prior to running our function, the user must also link the parallel and Rcpp libraries.

```
#bs function call
bs = function(x, degree=3, interior.knots=NULL,
              intercept=FALSE, Boundary.knots = c(0,1),
              type="serial", ncls=2)
{
    #input check
    if(missing(x))
    stop("You must provide x")
    if(degree < 1)
    stop("The spline degree must be at least 1")

    #input preprocessing
    Boundary.knots = sort(Boundary.knots)
    interior.knots.sorted = NULL
    if(!is.null(interior.knots))
        interior.knots.sorted = sort(interior.knots)

    #formation of knots
    knots = c(rep(Boundary.knots[1], (degree+1)),
                 interior.knots.sorted,
                 rep(Boundary.knots[2], (degree+1)))
    K = length(interior.knots) + degree + 1
        lenx = length(x)

    #parallel calls
        if(type == "openmp") {
                print("run type: openmp")
                dyn.load("bs_omp.so")
                Bmat = .Call("formMatrix",x,degree,knots,K,lenx)
        }
        else if (type=="cuda") {
                print("run type: cuda")
                dyn.load("bs_cuda.so")
                Bmat = .Call("formMatrix",x,degree,knots,K,lenx)
        }
        else if(type=="snow") {
                print("run type: snow")
                cls = makePSOCKcluster(rep("localhost", ncls))
                print(cls)
                Bmat = formMatrixSnow(cls, x, degree, knots, K)
        }
        else if(type=="serial") {
                print("run type: serial")
                Bmat = matrix(0,length(x),K)
                for(j in 1:K)
                        Bmat[,j] = basis(x, degree, j, knots)
        }
        else {
                print("Incorrect run type - serial used instead")
                #do serial
                Bmat = matrix(0,length(x),K)
                for(j in 1:K)
                        Bmat[,j] = basis(x, degree, j, knots)
        }
    #output postprocessing
    if(any(x == Boundary.knots[2]))
        Bmat[x == Boundary.knots[2], K] = 1
    if(intercept == FALSE)
        return(Bmat[,-1])
    else
        return(Bmat)
} #end bs function
```

A.2 Serial R

```
#Serial R code
#exploits Cox-de Boor recursion formula
Basis = function(x, degree, i, knots){
    #recursion base case
    if(degree == 0)
        B = ifelse((x>=knots[i])&(x<knots[i+1]),1,0)
    else{
        #alpha1 computation
        if((knots[degree+i]-knots[i]) == 0)
            alpha1 = 0
        else
            alpha1 = (x-knots[i])/
                     (knots[degree+i]-knots[i])
        #alpha2 computation
        if((knots[i+degree+1] - knots[i+1]) == 0)
            alpha2 = 0
        else
            alpha2 = (knots[i+degree+1]-x)/
                     (knots[i+degree+1]-knots[i+1])
        B = alpha1*basis(x,(degree-1),i,knots)+
            alpha2*basis(x,(degree-1),(i+1),knots)
    } #end else
    return(B)
} #end function
```

A.3 OpenMP

```
//C++ interface employing OpenMP
//project_omp.cpp

#include <Rcpp.h>
#include <omp.h>
using namespace Rcpp;

NumericVector basis(NumericVector x, int degree, int i,
                    NumericVector knots, int lenx)
{
    //memory allocation
    NumericVector B(lenx), alpha1(lenx), alpha2(lenx);
    if(degree==0) //recursion base case
        B = wrap(ifelse((x >= knots[i])
                 &(x < knots[i+1]), 1, 0));
    else{
        //alpha1 computation
        if((knots[degree+i] - knots[i]) == 0)
            alpha1 = rep(0,lenx);
        else
            alpha1 = wrap((x-knots[i])/
                          (knots[degree+i]-knots[i]));
        //alpha2 computation
        if((knots[i+degree+1] - knots[i+1]) == 0)
            alpha2 = rep(0,lenx);
        else
            alpha2 = wrap((knots[i+degree+1]-x)/
                        (knots[i+degree+1]-knots[i+1]));
        B = wrap(alpha1*basis(x,(degree-1),i,knots,lenx)+
                alpha2*basis(x,(degree-1),(i+1),knots,lenx));
    }//end else
    return B;
} //end basis function

RcppExport SEXP formMatrix(SEXP x_, SEXP degree_, SEXP knots_,
                           SEXP k_, SEXP lenx_)
{
    //convert data types from R to C++
    int degree = as<int>(degree_),
        k = as<int>(k_), lenx = as<int>(lenx_);
    //SEXP to NumericVector:
    //http://dirk.eddelbuettel.com/code/rcpp/Rcpp-quickref.pdf
    NumericVector x(x_);
    NumericVector knots(knots_);

    //output variable allocation:
    //matrix: lenx rows, k columns filled with 0
    NumericMatrix out(lenx,k);

    #pragma omp parallel for
    for(int j = 0; j < k; j++) {
        //R equivalent: for(j in 1:k)
        //Reference jth column; changes propagate matrix
        NumericMatrix::Column jvector = out(_,j);
        #pragma omp critical
        jvector = basis(x, degree, j, knots, lenx);
    }//end for(j)
    return out;
} //end matrix function
```

A.4 CUDA

```
#include <Rcpp.h>
#include <cuda.h>

using namespace Rcpp;

//CUDA kernel function: base case
__global__ void degree0(float *B, float *knots,
                        float *x, int i)
{//current thread
        int me = blockIdx.x * blockDim.x + threadIdx.x;
        B[me] = ((x[me] >= knots[i]) &&
                 (x[me] < knots[i+1])) ? 1 : 0;
}//end kernel

NumericVector basis(NumericVector x, int degree, int i,
                    NumericVector knots, int lenx)
{
    NumericVector B(lenx); // output variable
    NumericVector alpha1(lenx), alpha2(lenx);

    if(degree==0){
        //blocks and threads for GPU
        int nthreads = min(lenx, 500),
            nblocks = ceil(lenx/nthreads);
        //set up grid/block dimensions
        dim3 dimGrid(nblocks,1),
             dimBlock(nthreads,1,1);
        float *dB, *dknots, *dx;
        //copy to the GPU
        cudaMemcpy(dB,B,lenx,cudaMemcpyHostToDevice);
        cudaMemcpy(dknots, knots, knots.size(), cudaMemcpyHostToDevice);
        cudaMemcpy(dx, x, lenx, cudaMemcpyHostToDevice);
        degree0<<<dimGrid,dimBlock>>>(dB,dknots,dx,i);
        //copy kernel results back to B
        cudaMemcpy(B,dB,lenx,cudaMemcpyDeviceToHost);
        B = wrap(ifelse((x >= knots[i]) & (x < knots[i+1]), 1, 0));
    } //end if
    else{
        //alpha1 computation
        if((knots[degree+i] - knots[i]) == 0)
            alpha1 = rep(0,lenx);
        else
            alpha1 = wrap((x - knots[i])/(knots[degree+i] - knots[i]));
        //alpha2 computation
        if((knots[i+degree+1] - knots[i+1]) == 0)
            alpha2 = rep(0,lenx);
        else
            alpha2 = wrap((knots[i+degree+1] - x)/
                          (knots[i+degree+1] - knots[i+1]));
         B = (alpha1 * basis(x, (degree - 1), i, knots, lenx))
             +(alpha2 * basis(x, (degree - 1), (i + 1), knots, lenx));
    }//end else
    return B;
} //end basis function

RcppExport SEXP formMatrix(SEXP x_, SEXP degree_, SEXP knots_,
                           SEXP k_, SEXP lenx_)
```

```
{
    //convert data types from R to C++
    int degree = as<int>(degree_),
    k = as<int>(k_), lenx = as<int>(lenx_);
    //SEXP to NumericVector
    NumericVector x(x_);
    NumericVector knots(knots_);
    //output variable allocation:
    NumericMatrix out(lenx,k);
    //blocks and threads for GPU
    int nthreads = min(lenx, 500);
    int nblocks = ceil(lenx/nthreads);
    //set up grid/block dimensions
    dim3 dimGrid(nblocks,1), dimBlock(nthreads,1,1);
    float *dB, *dknots, *dx;
    //make space on GPU
    cudaMalloc((void **) &dB,lenx*sizeof(float));
    cudaMalloc((void **) &dknots, knots.size()*sizeof(float));
    cudaMalloc((void **) &dx, lenx*sizeof(float));
    for(int j = 0; j < k; j++) { //R equivalent:  for(j in 1:k){
        //Reference the jth column; changes propagate to matrix
        NumericMatrix::Column jvector = out(_,j);
        jvector = basis(x, degree, j, knots, lenx);
    }//end for(j)
    cudaFree(dB);
    cudaFree(dknots);
    cudaFree(dx);
    return out;
} //end matrix function
```

A.5 SNOW

```
#SNOW R code
#applies a cluster to each column of the final matrix
formMatrix = function(cls, x, degree, knots, K){
    sequence = 1:K
    #cluster function - returns column of the matrix
    basis = function(i,degree){
        #recursion base case
        if(degree == 0){
            B = ifelse((x>=knots[i])&(x<knots[i+1]),1,0)
        }
        else{
            #alpha1 computation
             if((knots[degree+i]-knots[i]) == 0)
                 alpha1 = 0
            else
                alpha1 = (x-knots[i])/
                         (knots[degree+i]-knots[i])
            #alpha2 computation
            if((knots[i+degree+1] - knots[i+1]) == 0)
                alpha2 = 0
            else
                alpha2 = (knots[i+degree+1]-x)/
                         (knots[i+degree+1]-knots[i+1])
            B = alpha1*basis(i,(degree-1))+
                alpha2*basis((i+1),(degree-1))
        } #end else
        return(B)
    } #end cluster function
    #have each cluster obtain a column of the matrix
    jvector = clusterApply(cls,sequence,basis,degree)
    #form the final matrix to return
    Bmat = Reduce(cbind,jvector)
}
```

B Contributions

The group met to complete the project in a mob programming like fashion. Each member contributed to the development of the code and the algorithms equally. The original concept of B-Spline regression was proposed by Brian Vargas, who also typed the LaTeXfile. Mark Fan, Michael Banzon, and Max Matsumoto wrote much of the write up as a word document which Brian later edited, converted, and completed the missing sections.