Parallelizing the Time Series Analysis of Phenometrics and Long-Term Vegetation Trends for the Great Plains Ecoregion

By: David Freeman, Jacob Peyton, Joe Boutte
For: Dr. Shawn Hutchinson
Concurrent Systems
Kansas State University

1. Introduction

The task being explored in this paper is how to speed up the processing of NDVI satellite images via an R package called BFAST by running it on the local HPC cluster, Beocat. The task stems from research Dr. Shawn Hutchinson is conducting into how human interaction impacts the grasslands of North America through the use of satellite data on how green the surface of the planet is. He then uses BFAST to detect trends and breaks in the data he receives.

The data that Dr. Hutchinson's research is based on comes from NDVI satellite imagery that is taken 23 times a year since 2001. Each NDVI picture, or tile, is of a unique 250m by 250m section of Earth's surface. He then takes these pictures and runs several scripts on them to obtain the data that is pertinent to his research. Specifically, how green each pixel of grassland in each tile has been has been since these pictures started being taken. This means that each pixel has 391 values from the last 17 years that these pictures have been taken. These "greeness" values get outputted into a comma separated value file with the pixel's ID number and all of its 391 values.

This is where Dr. Hutchinson runs the data through his BFAST R script and gets the trend and break analysis data his research needs. Since the only part he needs to be sped up is the processing of the data, that is where we step in. The plans on how to solve this problem differ only in how intensive they are to complete. The first plan is to manually parallelize by turning each tile into a different job in the Beocat queue. The second plan is to rewrite the R script using packages that exist to parallelize R. The third plan is rewrite the BFAST package and the R script in C.

2. Related Work

A group of researchers at the University of Copenhagen tackled exactly this problem earlier this year. In their paper titled "Massively-Parallel Break Detection for Satellite Data" they compared four different approaches to running tasks in BFAST. Their first method was simply to run BFAST in R, the way it was intended. They then followed up by rewriting the core algorithms of BFAST and running them in python. They then took their python code and ran it in parallel on a CPU. For their final comparison, they took their python code and ran it in parallel on a GPU. Ultimately, they were able to come up with an implementation that was four orders of magnitude faster than the standard R implementation of BFAST. This is beneficial for us, because it not only shows that the task of

parallelizing BFAST can be done, it also shows that rewriting the core aspects of BFAST can be done. It also shows that this could be parallelized on a GPU, which is not one of our main focuses currently, but is now opened up as an option in the future.

3. Implementation

a. Shell Script Parallelization

The first idea of how to parallelize the code is the most basic, most rudimentary, and least invasive to what resources we were given to start with. The plan of attack for this strategy is to simplify the process of submitting each job to the Beocat queue so each tile can be submitted to the Beocat queue in one, fell swoop and they can all run as the resources become available to do so. This method should net a speed up, maybe not the largest speed up, but a speed up nonetheless.

This method is the easiest on us as it requires almost no alteration to the R code, with the one alteration being in the first lines of the program to convert it to taking the filename as a command line argument. Changing that one part of the code allows it to remain untouched for the remainder of the development and testing of this method. The only other piece that this method requires is a C program that pulls a list of files in its directory and creates a shell file for each csv file that runs the program on the node it is assigned to. The C program also outputs one big shell script that contains a line for each csv file that submits all the jobs. The challenge here is to have the C program use the file size or number of lines of each tile to determine, with a safe margin, how long each job should run so time can allocated accurately. The problem this solves is that some of the tiles are under 100 kilobytes, some are over 700 megabytes, and the rest fall somewhere in between. The longer files will take close to a week to run while the smallest file takes about 10 minutes, the estimation is so that a job that will take 30 minutes doesn't get submitted with the scheduler being told to allocate 7 days to the task. Having each job submitted with a time allocation that is close to how long the job will actually take to complete allows the jobs to be placed as quickly as they possibly can be.

The potential benefits of this method is that the only changes that would need to be made for a new data set are to put them in the same directory as the C script, run it, and then run the produced shell script; which is simple and easy. Another benefit is that this method leaves the R script unchanged, which would allow Dr. Hutchinson to change it in the future if he ever needed to change which data was outputted from the program. The final benefit is that since each tile is its own separate job in the Beocat queue, if one fails Dr. Hutchinson simply has to go back and restart that one job that failed instead of having to start all of them again or have us build that kind of logic into a controller for a more parallel model.

The expected speed up with this method is tough to give an accurate calculation on as it depends on how the scheduler schedules the jobs. In the worst case, where each job runs serially on a single machine, the speed up is about 2 times faster than running on desktop machine. The best case, in which each of the 422 jobs start at the same time and run in what would be perfect, single-threaded parallel, the speed up would be about 20 times faster than if run on a desktop machine.

b. R Parallelization

The most straightforward approach to making a program written in R run in parallel is to take advantage of packages made explicitly for running R in parallel. This approach lies directly in the middle of all our possible solutions - both in terms of difficulty to implement and potential speed improvements.

To better understand how parallelization works in R, it is advantageous to first have a basic understanding of a function R uses to deal with looping through a repetitive task: lapply. Lapply takes two parameters: a function and a list of parameters that you want to pass to the function. Lapply then calls the function the same number of times as there are elements in the parameters list, passing a different element from the list each time. This all happens serially, but even just switching code over from a basic loop to lapply can garner significant speed increases due to the optimization in R.

R has a library called “parallel” that provides a few different functions that can be used to take the lapply function and run it in parallel. The first option is parLapply. ParLapply requires a bit of work to get set up and working such as creating a cluster and exporting any necessary functions and data to each of the cluster workers. However, there is another function, called mclapply, that does most of the work for you. There is no need to worry about setting up the environment for each of the cluster workers, because mclapply creates each worker process as a clone of the master right at the point it is called.

By taking advantage of one of the parallelization methods available directly in R, we can run every tile simultaneously. Therefor, without any other optimizations, the longest the job would take is the amount of time it takes to process a single tile. This can be further improved upon by breaking down the tiles into their individual data points, resulting in up to a potential 16,000,000x speed up. Of course, a job running 16,000,000 cores is completely impractical and the most optimal solution lies somewhere short of that massive number.

c. C Rewrite and Parallelization

R is a programming language written for statisticians. As such, it has a high level of abstraction compared to C, C++, and even Java or C#. The additional steps the compiler has to take to interpret the command-line based source code can massively slow down the program, compared to an equivalent program written in

C. An example of how massive this slow-down is, a few economists out of University of Maryland and University of Pennsylvania did some benchmarking with various programming languages. Their benchmarking showed R running 300-500 times slower than an equivalent C++ program (Aruoba and Fernandez-Villaverde 2014). If Dr. Hutchinson's code and the BFast library he uses can be ported over to C, we can expect at least a ~250x speed-up in runtime before we start parallelizing the code.

Being able to rewrite all of that code in C is a difficult task, but it is possible. R itself is written in C and Fortran, so any in-built functions used by Dr. Hutchinson's code or BFast can be written in C without issue. BFast itself is open source, available in a repository on the R-Forge website. Additionally, the entirety of the package appears to be only 14 .R files, the longest only being around 150 lines long. So deciphering the BFast package is merely a matter of following the function calls in those 14 files.

If we are able to rewrite the entire program in C within the given timeframe, we will also attempt to parallelize it. The time remaining on our project will likely determine which method we use. Pthreads and OpenMP are the two C parallelizing extensions we are currently familiar with, and they will be the first we utilize. Pthreads will be the favorite if we are short on time, being incredibly simple to implement, while OpenMP may provide a faster speed-up if we manage to optimize the package delivery across threads. Since this project is 'perfectly parallelizable', the speed-up from parallelizing the C code is expected to be slightly less than linear; running it simultaneously on 2 cores will be about a 2x speed-up, running on 16 will be about a 16x speed-up, and so on. The program likely won't have a perfectly linear speed-up, due to some overhead from the parent node, but it should still be close to linear. With that said, running this program on 64 cores after rewriting it in C should hopefully get us a 15000x speed-up, compared to running it serially, potentially reducing the run-time from months to minutes.

4. Evaluations

We only have two criteria on which plans to prioritize. Dr. Hutchinson's one criteria on this project is that he is able to understand the code and make changes to it in the future. He is already familiar with R and shell scripts, so Plans A & B are preferred. However, he has allowed us to try to rewrite it in C, but only if he is able to understand it. This would require a lot of commenting, and possibly some face-to-face explanation of the code, but it's not impossible, so Plan C is not ruled out. On the other hand, Dr. Andresen's one criteria is to simply make the program run as fast as computationally possible, and have it be done before the end of the semester. From that criteria, we would prefer Plan C, if we were sure we could complete it in the given time frame. So, the current plan of attack is to complete either Plan A, Plan B, or both, which likely won't take the rest of the semester, and then attempt to complete Plan C in the remaining time.

Once we have implemented at least one of these plans, we will perform some run-time analysis. Our main goal for this project is to simply try to make it as fast as possible, whatever means necessary. So, we will be looking at time spent in different function calls, time required to process the files on Beocat, time spent writing to files, etc., and we will be trying to optimize them as best we can. The plan that results in the fastest runtime will be given over to Dr. Hutchinson, and submitted to Dr. Andresen.

5. Conclusion

At this time, without having any of the plans fully completed and no testing done, we expect good results. The words “speed” and “up” were not used in that sentence because at the moment there is no guarantee that there will be any speed up. Plans A and B should improve the time over what Dr. Hutchinson currently gets when running all of the data on his desktop, simply because it is running on a server grade processor with no other threads using up resources, as in also running an operating system. That said, there should be some speed up just because the jobs won’t be getting run serially, as he has been doing from the beginning, but with the possibility to be distributed across multiple threads.

While plans A and B are certain to work on their own, the big goal will be plan C, aptly named because it seeks to rewrite everything in C. This plan should be the fastest out of them all because of the speed which inherently comes from using C over R, but comes with the hindrance of non-computer people having difficulty understanding it.

With all of this in mind, it is looking good for Dr. Hutchinson’s research as his run time for producing figures with each new data set should be down from 5 months as he does it currently, to about 1 month with plans A and B, and maybe even less than a week with Plan C, but that requires us to finish plans A and B first.

Works Cited

Aruoba, S. Boragan, and Jesus Fernandez-Villaverde. *A Comparison of Programming Languages in Economics*. University of Maryland, 5 Aug. 2014, www.sas.upenn.edu/~jesusfv/comparison_languages.pdf.

Jones, Matt. *Quick Intro to Parallel Computing in R*. 25 July 2017, nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html.

von Mehren, Malte, et al. *Massively-Parallel Break Detection for Satellite Data*. University of Copenhagen, 4 July 2018, arxiv.org/pdf/1807.01751.pdf .

Weston, Steve. “Understanding the Differences Between Mclapply and ParLapply in R.” *Stackoverflow*, 19 July 2013, stackoverflow.com/questions/17196261/understanding-the-differences-between-mclapply-and-parlapply-in-r.

Parallelizing the Time Series Analysis of Phenometrics and Long-Term Vegetation Trends for the Great Plains Ecoregion

David, Jacob, and Joe

Satellite Imaging Project

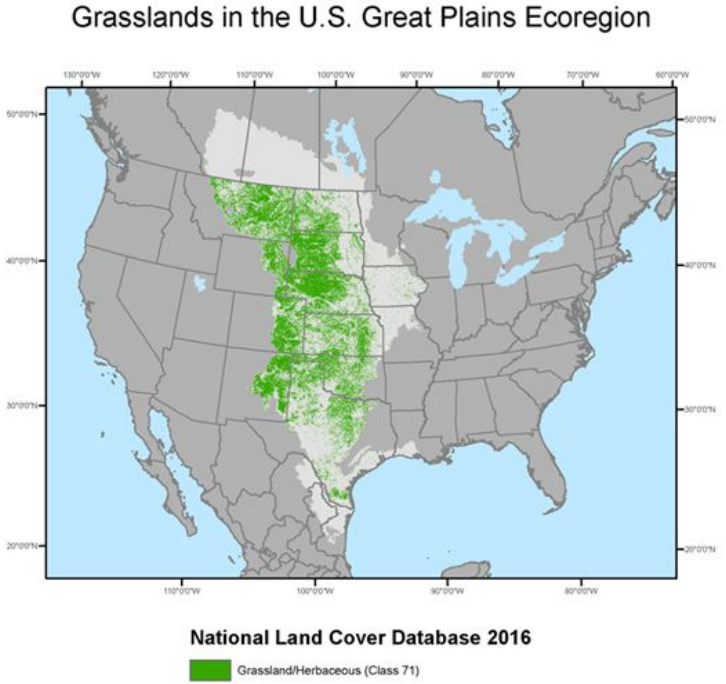
David, Jacob, and Joe

Goals

- ➔ Process a time series of MOD13Q1 images to assess trends in grassland biomass production as measured with NDVI/EVI. Develop and approach to identify “anomalous” digital number values for this proxy to inform near real-time management.
- ➔ Process time series of MOD13Q1 images to assess grassland phenology. Extract multiple phenometrics and assess their temporal trends and spatial variation over the period 2001-2017.
- ➔ Develop a validation procedure for Objectives 1-2 using imagery from the PhenoCam Network.
- ➔ Using an improved understanding of grassland development and production in the past, and forecasted into the future, assess the impact of changing grassland conditions on the provision of a subset of ecosystem services.

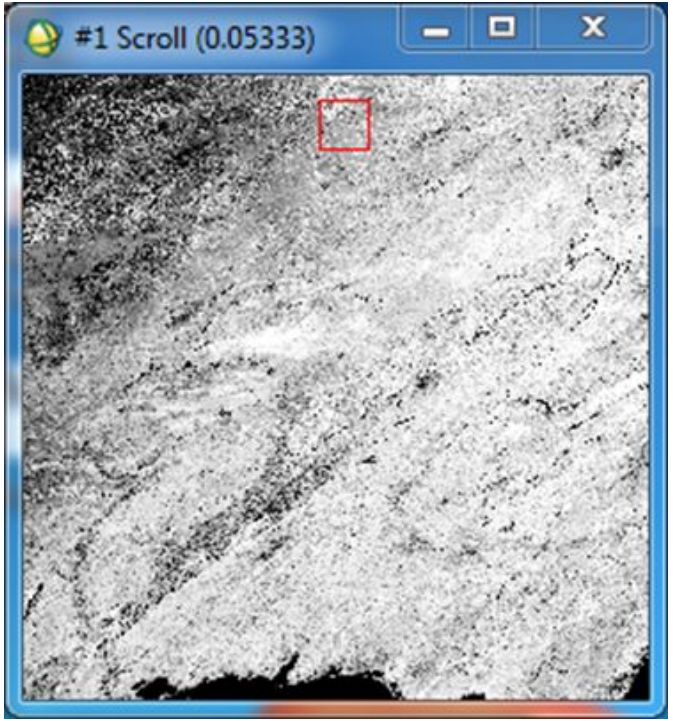
Goals TL;DR

- ➔ Process the fancy satellite images to find a trend in how green the 15.5 acres it represents was
- ➔ Process the same fancy images but this time we are trying to track the growth stages of the plants
- ➔ Use the data from both 1 and 2 to help make sure ground readings are correct
- ➔ Use all this data and forecast it to see how we impact the grasslands



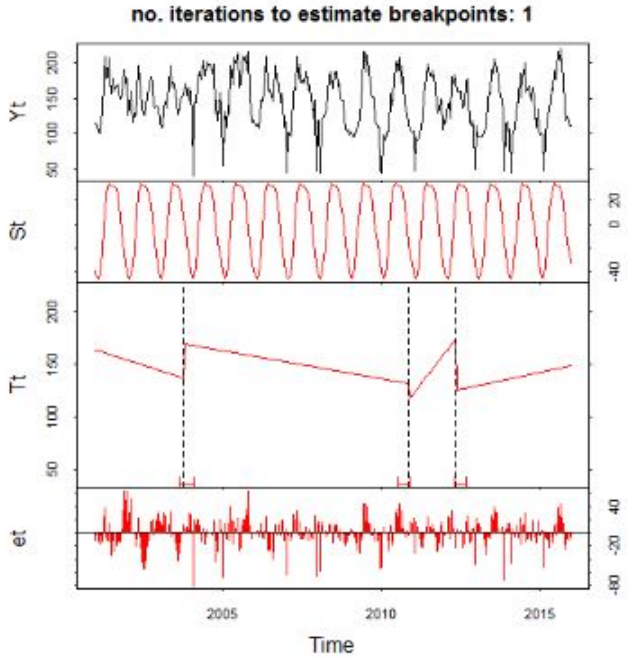
About those fancy space pictures

- There are 422 tiles, or pixels, each of which contain values for a separate 250m x 250m area
  - This is 15.5 acres for anyone who cared about converting
    - 6.2 Hectares, if anyone is still counting
- Since 2001, there have been 23 images a year for each tile
- This means we have 391 values for every data point measured in each tile
- TL;DR is that there is consistent and reliable data over a long period of time



The Computer Part

- Our task is to use an R package called BFAST
  - Stands for Breaks For Additive Season and Trend
- This is the black box we have to run the data through
  - And hopefully make this part faster



Methods of Attack

- ➔ Plan A - **Easy** - Manual Parallelization through scripts that allow all of the tiles to be submitted as jobs at the same time and allowing the scheduler to do the work
- ➔ Plan B - **Medium** - Parallelize the R code using some of its built in functions
- ➔ Plan C - **Hard** - Rewrite the whole thing, including BFast, in C, because we're masochists

Plan A: Manual (Shell Script) Parallelization

- ➔ The plan is to make a C program that generates a shell script for all the tile data files in the folder with the executable
- ➔ The shell script then gets run which submits jobs for all 422 tile data files using their size to safely estimate how much time each should take to run
- ➔ This allows him to run 2 things and have all of his data in Beocat waiting for the resources to run and not requiring him to run each jobs serially

Plan A: Manual (Shell Script) Parallelization

- ➔ Some benefits of this method are that the R code is already written and minimal effort was required to have it take the filename as an argument
  - ◆ From this: “filename <- “tile_###”
  - ◆ To this: “args <- commandArgs(trailingOnly = TRUE); filename <- args[1]”
- ➔ Another is that since each tile has its own job in the Beocat queue, if one fails he can re-submit only the failed job
- ➔ The last benefit is that once the code is written, all he has to do is drag in the new tile data files and run the 2 files again
- ➔ The length the code will take to run ranges from his original time of 5 months to the best case scenario of what should be under a week because every job would run at the same time and the total time would be however long the longest job takes
- ➔ This means my speedup will be between 1 and 21
  - ◆ Otherwise, it depends, unfortunately not on caching and locality

Plan B: Common Sense (R) Parallelization

R has a function called lapply that takes in two parameters: a list and a function. Similar to a foreach, lapply calls the function once for each item in the list, passing a different item as the parameter in the function each time. This is done serially.

```
lapply(1:3, function(x) c(x, x^2, x^3))
```

```
[[1]]
[1] 1 1 1
[[2]]
[1] 2 4 8
[[3]]
[1] 3 9 27
```

Plan B: Common Sense (R) Parallelization

R has a package specifically for parallelization called... parallel (duh). This package can be used to basically lapply in parallel. The parallel version of lapply is called parLapply and takes in an additional argument, the number of cores.

```
library(parallel)
cluster <- makeCluster(#ofCores)
parLapply(cluster, 2:4, function(exponent) 2^exponent)
```

```
[[1]]
[1] 4
[[2]]
[1] 8
[[3]]
[1] 16
```

Plan C: Insane (Rewrite in C) Parallelization

- ➔ BFast is Open Source
- ➔ Made up of 14 .R files, each 10-150 lines
- ➔ R is Compiled from C & Fortran
- ➔ Potential 500x Speed-Up without Parallelizing

```
bfast <- function(Yt, h=0.15, season =c("dummy","harmonic","none"), max.iter = NULL, b
{
    season <- match.arg(season)
        level = rep(level, length.out = 2)
    ti <- time(Yt)
    f <- frequency(Yt)      # on cycle every f time points (seasonal cycle)
    if(class(Yt)!="ts")
        stop ("Not a time series object")
    ## return value
    output <- list()
    Tt <- 0

    # seasonal model setup
    if (season=="harmonic") {
        w <- 1/f # f = 23 when freq=23 :-)
        tl <- 1:length(Yt)
        co <- cos(2*pi*tl*w); si <- sin(2*pi*tl*w)
        co2 <- cos(2*pi*tl*w*2);si2 <- sin(2*pi*tl*w*2)
        co3 <- cos(2*pi*tl*w*3);si3 <- sin(2*pi*tl*w*3)
        smod <- Wt ~ co+si+co2+si2+co3+si3
        # Start the iterative procedure and for first iteration St=decompose result
        St <- stl(Yt, "periodic")$time.series[, "seasonal"]

    } else if (season=="dummy") {
        # Start the iterative procedure and for first iteration St=decompose result
        St <- stl(Yt, "periodic")$time.series[, "seasonal"]
        D <- seasonaldummy(Yt)
        D[rowSums(D) == 0,] <- -1
        smod <- Wt ~ -1 + D
    } else if (season == "none") {
        print("No seasonal model will be fitted!")
        St <- 0
    } else stop("Not a correct seasonal model is selected ('harmonic' or 'dummy') ")

    # number/timing of structural breaks in the trend/seasonal component
    Vt.bp <- 0
    Wt.bp <- 0
    CheckTimeTt <- 1
```

Choosing a Plan

Dr. Hutchinson’s One Requirement:

He can read and operate it -> Plan A/B

Dr. Andresen’s More Important Requirement:

It has to be as fast as possible -> Plan C

Questions?

Sources

- http://gforge.se/2015/02/how-to-go-parallel-in-r-basics-tips/
- http://dept.stat.lsa.umich.edu/~jerrick/courses/stat701/notes/parallel.html
- https://www.rdocumentation.org/packages/bfast/versions/1.5.7/topics/bfast
- https://cran.r-project.org/web/packages/bfast/bfast.pdf
- https://www.sas.upenn.edu/~jesusfv/comparison_languages.pdf
- https://r-forge.r-project.org/scm/viewvc.php/?root=bfast