Scalability of Land Use Monitoring Codes

Rishu Saxena and Layne T. Watson
Departments of Computer Science, Mathematics, and Aerospace and Ocean Engineering
Virginia Polytechnic Institute and State University, Blacksburg, VA 24060

Randolph H. Wynne and Valerie A. Thomas
Department of Forest Resources and Environmental Conservation
Virginia Polytechnic Institute and State University, Blacksburg, VA 24060

Abstract. Earth observation satellites (EOS) such as Landsat provide image datasets that can be immensely useful in numerous application domains, by extracting information via time series analysis (TSA). While several TSA algorithms exist, EOS datasets are prohibitively large, currently of the order of petabytes and growing. High performance computing (HPC) resources and optimized, scalable implementations, therefore, are indispensable. This paper presents implementations of two TSA algorithms in two languages — Python, an interpreted language using which much scalable computing is recently being achieved, and Fortran, a compiled language. Scalability and performance results are presented. In addition, a brief assessment on choice of programming language is also discussed. This work will be directly useful in land use and land cover change studies, of interest to terrestrial science research, especially regarding anthropogenic impacts on the environment, and in much broader applications such as health monitoring and urban transportation.

Keywords. Time series analysis, big data, change detection, scalable computing.

1. Introduction

Land use and land cover change (LULCC) is of crucial importance globally [8], [9]. With anthropogenic activities such as deforestation and urbanization increasing exponentially through the past century, there have been significant changes in land cover in several parts of the world [7]. Simultaneously, significant changes in the global climate have also been observed, driven in part by LULCC [6]. LULCC also has significant impacts on a wide variety of other ecosystem services. Much research is, therefore, being directed towards Earth monitoring.

Earth observation satellites (EOS) such as Landsat provide image datasets that, if harnessed well, can be immensely beneficial towards understanding and managing our natural resources and LULCC monitoring. An excellent way of analyzing these satellite images for LULCC studies is time series analysis (TSA). For TSA, several images of the scene under consideration, taken over a period of time, are stacked together chronologically, and are subsequently analyzed. Commonly, the time series for each pixel is treated individually; the full image stack is thus a collection of many time series. The objective is to discover a ‘trend’ in how different relevant variables (indicators) evolve over time. Analysis made is based on the behaviors of the time series of these variables. When the trajectory of one or more of the variables departs from the normal, a change is detected. TSA for LULCC studies has been receiving increasing attention in the remote sensing community in the last decade, after the Landsat data became freely accessible in 2008 ([16]). Several time series analysis algorithms have been proposed by different groups in the community.

While several TSA algorithms exist, the size of the datasets itself is prohibitive, currently of the order of petabytes and growing. An EOS image stack typically consists of multiple images of a fixed area on the Earth’s surface (same latitudes and longitudes) taken at different time points. Experiments on multicore servers indicate that carrying out meaningful time series analysis on a single interannual, multitemporal stack with existing state of the art codes can take several days. An HPC platform for time series analysis of satellite images obtained from MODIS (Moderate resolution imaging spectroradiometer) was presented in [2]. In contrast with Landsat images, MODIS images have much coarser spatial resolution (250 to 1000 meters compared to Landsat’s 15 to 60 meters) but much better temporal resolution (1 to 2 days compared to Landsat’s 16 days). Several architectures such as massive parallel clusters, heterogeneous clusters, grid computing, GPUs and the like have also been proposed [12].

This paper presents the implementation of two currently existing LULCC algorithms in Python 2.7 and Fortran 2003. Parallelization across pixels is implemented and further possibilities for speeding up the implementations are discussed. For a stack of 198 images, each having 10^6 pixels, the EWMACD Python implementation exhibits a speedup of 49 and EWMACD Fortran code a speedup of 47 using 64 cores. Optimized Fortran sequential codes for EWMACD are ≈ 28 times faster than optimized Python codes, though. Further, scalability results for processing
an image with approximately $10^8$ pixels using 64 cores are presented. A brief qualitative and quantitative assessment (scalability, performance, development time) of Python codes vs. Fortran codes is discussed. This work is a case study in the tradeoffs across the two languages in building high performance codes for new application domains.

Section 2 describes the two algorithms qualitatively. Experimental results are presented in Section 3. Section 4 describes optimizations and parallel implementations for each language, followed by a description of the experience with and a summary of the pros and cons of each language. Conclusions are presented in Section 5.

2. Change Detection Algorithms

Exponentially Weighted Moving Average Change Detection (EWMACD). Originally proposed in [3], [4], this algorithm models the time series as a trigonometric polynomial. The main steps in the algorithm are as follows: (i) Harmonic regression coefficients are computed using the data collected in the initial (two or more) years. For the full time series, this initial fit is used to get $\bar{I}$, a set of ‘good indices’ (data points with prediction error $E$ less than a threshold). The outlier free training data is used to obtain improved harmonic regression coefficients $\alpha = (\alpha_0, \ldots, \alpha_{2K+1})$, to be used for the rest of the processing. Here $K$ is the number of harmonics to be used. (ii) EWMA of the residuals is calculated as $z_i = E_{\tilde{j}_i} + (1 - \lambda)z_{i-1} + \lambda E_{\tilde{j}_i}$, $i = 2, \ldots, |\bar{I}|$, $\tilde{j}_i \in \bar{I}$. Here, $\lambda \in (0, 1]$ is the weight given to the most recent residual. (iii) A control limit vector $\tau$ is developed based on the standard deviation of the outlier free training data residuals, and $\lambda$. (iv) EWMA flag history $f$ is now developed using $z$ and $\tau$: $f_s = \text{sgn}(z_s)[|z_s/\tau_s|]$, $\forall s = j_i \in \bar{I}$. $f_s = 0$, otherwise. Finally, if there is a run of $+1$ or $-1$ in the values $\text{sgn}(\Delta f_s) = \text{sgn}(f_{s+1} - f_s)$ of length $\varpi$, called the ‘persistence’, a change is signalled at the index $s$ beginning the (nonzero) run. The training period, $K$, $\lambda$, $\varpi$, the thresholds are all parameters of the algorithm. EWMACD is able to capture seasonal changes. The performance of EWMACD is yet to be studied for situations where the training data is not sufficient or where some change is already a part of the training data.

Breaks For Additive and Seasonal Trend (BFAST). BFAST decomposes the given time series iteratively into three components: trend, seasonal, and noise. BFAST computes and evaluates least squares fits in windows of increasing size. Qualitatively, (i) first the possibility of there being any structural change in the given time series is determined by computing the partial sums of recursive residuals of least squares fits in windows (OLS-MOSUM). The limiting process of these partial sums is the increments of a Brownian bridge process [5]. If the observations do have a structural change, an ordinary linear least squares fit will result in large recursive residuals and, hence, in large partial sums. Therefore, the occurrence of large values in the process is an indication of the presence of a structural change — this probability being calculated from the Brownian bridge table. (ii) If a structural change is indicated, a search for change location is done. Each interior time point $t$ is considered a breakpoint (change location) candidate. A recursive residual is the error at time $t_j$ from the linear least squares fit over the window $[t_i, \ldots, t_j]$. The breakpoints (change locations) are chosen so as to minimize the sum of squared recursive residuals over all windows in between (omitting) the breakpoints. This is done for both trend and seasonal components of the time series, in each iteration. Overall, BFAST is a computationally intensive bottom-up approach. While the experiments so far assure that the method captures the linear trend correctly, it’s ability to capture phenological (seasonal) changes has not been studied sufficiently yet. BFAST was originally proposed in [15].

3. Experiments

The two algorithms are implemented in Python 2.7 and Fortran 2003. Python is an interpreted language, being increasingly used for scientific computing, with avenues to improve its performance for scientific computing being explored. Fortran is a compiled language.

Input data. Results for a study area located in South Carolina are presented here. Figure 1 shows the satellite images taken from Landsat path 18, row 37, on dates January 3rd, 2009, and February 16th 2014, corresponding to the beginning and end of the image stack (henceforth referred to as SC1837) under consideration, which has 198 time points and one band, the normalized difference vegetation index [11] $NDVI = (\text{NIR} - \text{VIS})/(\text{NIR} + \text{VIS})$, where NIR is the near infrared (band 4, biomass) and VIS is the visual red (band 3, vegetation slopes). NDVI is known to be a good metric for vegetation cover, where negative values of NDVI are deemed irrelevant as they correspond to water, clouds, or missing observations. For processing, positive values of NDVI are scaled by 10,000 and negative values are masked out (set to $-9,999$). The image stack dimensions are $R = 7411$, $C = 8801$, $S = 198$, $B = 1$, where $R, C, S$, and $B$ denote the total number of rows, columns, time points, and spectral bands in the stack, respectively. Like most time series analysis algorithms, both algorithms rely on user defined parameters, requiring, in general, a priori knowledge of the scene. The experiments here adhere to the published values of the parameters for each of these algorithms. Specifically, EWMACD utilizes 2 harmonics, $L = 3$, $\lambda = 0.5$ for control chart parameters, and persistence equal to 7. All the time points in the years 2001 to 2003
are used as training data. BFAST utilizes 1 harmonic, probability threshold of 0.7 to determine the presence of structural change (via Brownian Bridge); a maximum of two iterations is allowed, the algorithm searches for two break points, and the minimum spacing between consecutive break points is set to 5% of total available time points.

**Validation.** National land cover database (NLCD) and tree canopy cover (TCC) data are used for checking the accuracy of algorithms outputs in this work. NLCD is, by far, the most accurate data available for land cover classification in the U.S. NLCD classification maps are available for the years 2001, 2006, and 2011. TCC data [14] is also available for specific years. For a 30m × 30m area (one pixel), TCC is defined as the proportion of the area that is covered by tree canopy versus “not tree canopy”. Methods to measure the TCC for a pixel are known.

One dimensional results for two pixels are displayed in Figure 2. Per NLCD classification, both these pixels are forests in 2001. They both lose foliage and are predominantly covered with scrubs/shrubs by the year 2011. TCC for the pixel in Figure 2(top) reduces from 93.6% in 2009 to 87.2% by 2013, an approximately 6.4% loss. The NDVI trajectory indicates a sharp loss in 2007 and then an unsteady but increasing recovery from 2010 onwards. Both EWMACD and BFAST capture the overall changes correctly, including the 2007 loss and the increasing NDVI from 2009 onwards.

The pixel displayed in Figure 2(bottom) lost approximately 45% in TCC between 2009 and 2013. The NDVI values, in general, show a sharp drop in mean in 2003 followed by an increase in mean starting in 2004. BFAST accurately captures the trend, the sharp loss in 2003 and the overall tendency to recover 2004 onwards. EWMACD indicates stability more or less throughout the 12 year time span, indicating only minor disturbances around 2005. This can be attributed to ill-chosen parameter values for this pixel. In addition, both algorithms fail to
indicate the TCC loss that occurred during the last four year time period. This is potentially because this loss is not evident from the NDVI trajectory itself — indicating that NDVI alone may not be sufficient for land cover monitoring; more bands might need to be considered.

The full image stack SC1837 is processed for the time period 2009–2013. The images in this stack have $7411 \times 8801 = 65224211 > 10^7$ pixels. The sequential baseline implementation of BFAST in Python (faster than that in Python) takes over 7 hours to analyze a $1000 \times 1000 = 10^6$ pixel subset of the full scene. For EWMACD, its sequential baseline implementation in Python takes about an hour for the $10^6$ pixels subset. Fortran sequential baseline implementation takes 2 minutes, 18 seconds for this subset. For EWMACD, thus, Fortran is apparently 26 times faster than Python. Of the sequential baseline implementations, only Fortran EWMACD code is used for processing the full image stack (full scene, 198 time points). It takes 1 hour and 20 minutes to complete. (Details of optimizations used are included in section 4.) Figure 3 displays EWMACD results on a few (2) time points. At any given time point, a black pixel indicates being flagged by EWMACD as having no disturbance, a green pixel as in recovery, and a red pixel as in loss. So an area to the northeast of pixel $(4500, 4000)$ was in vegetation loss in September 2013 while by February 2014, the entire area to the right of $(4000, :)$ had substantially recovered.

4. SCALABILITY AND PARALLELIZATION

Infeasibility of serial baseline implementation of BFAST for a full scene was alluded to in Section 3, last paragraph. EWMACD is faster but still takes, at best, over an hour to process $10^7$ pixels. Knowing that (i) the image stack discussed in this paper consists of only 198 time points (2009–2014) while there are currently 900 time points (1984–2014) actually available, (ii) the run times discussed here are for a single path/row only while there are more than 450 path/rows in the US alone, and other similar facts, scaling the codes is imperative for any meaningful analysis. This section presents (i) the optimizations used to address the computational hotspots and hardware bottlenecks, and (ii) the parallel implementations for both Python and Fortran. The overall development experience is summarized in the end.

First, since both algorithms involve a fair number of arrays, attention is given to the creation and use of array variables. Second, a major concern in designing parallel codes for Landsat imagery is load balancing across pixels. Specifically, since the time series processing for any given pixel is independent of that for any other pixel, the algorithms are apparently embarrassingly parallel with respect to pixels. Landsat images, however, suffer from missing observations (due to factors beyond human control), thereby resulting in ‘invalid’ pixels (a pixel is declared invalid if there are fewer than $2K + 1$ observations in the entire time span, where $K$ signifies the number of harmonics being used for the EWMACD). These invalid pixels are nonuniformly distributed across the data, which induces very high work load imbalance across the pixels.

4.1. Python

Arrays in Python are handled using NumPy, high level inbuilt support for working with multidimensional arrays. NumPy automatically vectorizes array operations making them very efficient. Together with SciPy, the fundamental scientific computing library of Python, Numpy supports most linear algebra operations, a feature of interest with regard to the algorithms considered here. In addition, Python also supports creation of anonymous functions using a keyword ‘lambda’. This allows direct array to array transformations, preserving the efficiency of vector instructions. For example, $\sin(jt_i), i = 0, \ldots, M$ may be carried out as $\sin(map(lambda x : x \ast j, t[0 : M]))$. For the algorithms considered here, not all array operations are
directly portable to Numpy; lambda expressions facilitate the porting in these instances. Overall, significant gain in performance over the baseline implementation is noticed on replacing Python primitives with NumPy primitives. For EWMACD, approximately 25% of the total time originally went in determining the validity of a pixel, and 20% in least squares fitting, 20% in calculation of residuals. On replacing the Python primitives with NumPy primitives, these numbers reduce to 0.7%, 16.5%, and 15.4%, respectively. Other procedures such as computing the control charts, flagging history gain prominence as computational hot spots, each one taking 16% of the total time. Unlike least squares fitting and residual calculations, this latter set of procedures is not expressed as concise mathematical operations, NumPy cannot be utilized for their optimization.

Python sequential codes are (weakly) parallelized using the multiprocessing module available in Python. Multiprocessing creates subprocesses with disjoint virtual memory space. The image stack is chunked pixelwise and these chunks are distributed to the processes. Within a subprocess, that chunk is processed sequentially. The full scene is processed using 64 subprocesses in approximately 64 minutes. For this full scene run, a load imbalance is also noticed across subprocesses, because the chunking of pixels utilized here does not ensure load balancing. An improved way to distribute chunks would be to actually partition the image into a larger number of chunks (for example, 3000 chunks) and then distribute these chunks across the subprocesses nonuniformly. However, this is currently not implemented. Further, Python is an interpreted language with a global interpreter lock (GIL) (https://wiki.python.org/moin/GlobalInterpreterLock). To bypass GIL enforced serialization and run multiple threads in parallel without compromising thread safety requires nontrivial programming time and is still risky. With the current implementation and the run times recorded here, processing the full scene using Python EWMACD code with a single process will take more than a day and is, therefore, not attempted.

4.2. Fortran

Fortran has intrinsic support for high dimensional arrays. The algorithms implemented here necessitate the use of several intermediate (work) arrays. Since allocation/deallocation of arrays is expensive, global arrays are used for all work arrays. Fortran vector instructions are utilized wherever possible. LAPACK and BLAS libraries are used for all linear algebra operations.

Fortran sequential codes are parallelized using OpenMP. In contrast with Python’s multiprocessing, OpenMP is a shared memory paradigm. The input and output arrays are the only large arrays; these are declared as shared variables. The remaining significant hardware bottleneck is the load imbalance across pixels.

Attempting to weed out invalid pixels in a preprocessing step and execute the PARALLEL DO loop for only valid pixels leads to CPU underutilization (from 99.9% to 70–80%), simultaneously increasing the OpenMP time. This presumably is due to memory contention: the memory access pattern for the latter approach is such that multiple threads try to access the same memory bank(s). Furthermore, even amongst the valid pixels, the total number of observations available for one pixel can be much less than the number of observations available for some other pixel. So, even with this preprocessing approach the work load balance is not guaranteed.

Finally, allocating/deallocating arrays within each thread is inefficient. Allocatable global arrays in modules can be used by threads via THREADPRIVATE, but this data copy mechanism does not work with dynamic loop scheduling, which is desirable because of the large variance in pixel analysis times (including missing data for a pixel). The best alternative is using Fortran automatic arrays with OpenMP PRIVATE.

The computational hot spots for Fortran codes are identified. For EWMACD, more than 50% of the time is spent in least squares fitting, specifically in DGELS (LAPACK) calls. LAPACK is already optimized for the hardware. 22% of the total time is in the calculation of residuals. This subroutine has two DO loops with dependencies and cannot be vectorized. For BFAST, approximately 97% of the OpenMP time is spent in computing the recursive residuals. Essentially, linear and harmonic least squares fits are done in every permissible interval, and the least squares fitting is already optimized.

In summary, after considering and testing several alternatives, the best approach found was to (1) perform the raw binary stream input data order \((r, c, s, b)\) conversion to \((s, c, r, b)\) order in parallel; (2) cull invalid (including missing) pixels inside the subroutines EWMACD and BFAST, which are called from within a PARALLEL DO; (3) convert the nested DO loop \(D_0 r=1,R ; D_0 c=1,C\) into a single PARALLEL DO loop \(D_0 k=1,R*C,A\); (4) use OpenMP SCHEDULE(DYNAMIC,1); (5) process a chunk of \(A\) pixels indexed by \(k\) on each call to EWMACD and BFAST; (6) use automatic rather than allocatable arrays for all small work arrays in the subroutines, and allocate/deallocate just one large work array in both EWMACD and BFAST; (7) perform all I/O outside parallel OpenMP constructs to reduce memory and disk contention. Note that manually collapsing the nested loops and chunking within the pixel processing subroutine is more efficient than collapsing and chunking at the OpenMP directive level, since the latter would call the subroutines, which allocate and deallocate numerous work arrays, for each pixel index. The difficulty of
load balancing “embarrassingly parallel” applications is analyzed theoretically in [1].

Figure 4(top) shows the scaled speedup for BFAST, i.e., increasing both the problem size and the number of cores. The isoefficiency (constant efficiency as both the problem size and number of cores are scaled up) decreases significantly, indicating some combination of poor load balancing (the pixel chunk size $A = 100$), main memory contention, and increasing thread management overhead, as yet unresolved.

Parallel results for the full scene, processed only with EWMACD, are shown in Figure 4(bottom). Using 64 cores, the full scene is processed in 1.86 minutes, with a speedup of roughly 46x. The input binary image is 25GB. For this image, the code needs 74GB of memory. When a single thread (core) is used, the cache miss rate is 0.566% and 1.03 instructions per cycle are executed. For 64 cores, the cache miss rate is 0.721% and 0.52 instructions per cycle are executed. On 64 cores, the FLOPS performance of the EWMACD code is 48.54 GFLOPs. The peak theoretical performance for this machine is 358.4 GFLOPs, yielding performance to peak ratio of $48.54/358.4 = 13.54\%$.

For all the codes, the input (as well as output) image stacks are in binary file format. The logical mathematical description of an image stack uses the index order $(r, c, s, b)$, where $r, c, s$, and $b$ denote the row, column, time point, and spectral band, respectively. Python accesses arrays in row order. However, chunking of pixels for multiprocessing requires all the information for a given pixel to be contiguous. Without this, the data cannot be correctly chunked. Fortran accesses array elements in column order. To ensure good memory locality and due to the hardware effects of cache misses and paging, the image stack in Fortran implementation must be stored and processed in the index order $(s, c, r, b)$.

Failure to use this latter index order can result in a cache miss rate as high as 28%. So for both Python and Fortran, the input image is stored in time sequential format.

In python codes, Numpy is used for loading, reading and writing binary files. In Fortran codes, Fortran I/O with streaming access is utilized to read and write these files. Kernprof was used for profiling Python codes, valgrind for Fortran. Perf was used to determine cache misses. Visualizations are done in Python. The results presented in this paper were obtained on a single machine: 64 core AMD Opteron 6276, 1.4GHz CPU, 2MB cache per core, 265 GB main memory, CentOS. Python 2.7 and gfortran compiler version 4.8 were used for respective languages.

### Python vs Fortran performance

For a $10^6$ pixels scene with 198 time points, EWMACD Python code takes 68 minutes to complete serially and 1.38 minutes using 64 processes, exhibiting a speedup of approximately 49; EWMACD Fortran code takes 138.43 seconds serially and 2.96 seconds using 64 threads, a speedup of 47. The speedups are similar but the actual execution times are vastly different for the two languages: Fortran codes are 28 times faster than their Python counterparts. Further, in going from 32 to 64 processes, the performance gain for Python EWMACD is only 23% while Fortran EWMACD gains 40% (ideal would be 50%). This indicates weaker cache utilization by Python. An overall 7% cache miss rate is observed for Python codes after all optimizations; for Fortran it is 0.72%. For a $64M$ pixels image, Fortran parallel code is 35 times faster than Python parallel code, possibly due to lack of load balancing in Python. Finally, processing this $10^6$ pixel scene using EWMACD serial codes written in R, another language popularly used in remote sensing community, takes about 140 minutes. Serial Fortran codes for EWMACD have thus yielded a speedup of 50 over serial R codes.

### Development life cycle

The algorithms presented here exist in the literature but their concise mathematical descriptions are not published. Developing pseudocodes from the published codes took nearly a month each. Most of the time for BFAST went in reverse engineering
parts of BFAST codes to get insight into the original implementation (written in R). Most array operations are simple one line instructions in Python (with NumPy and SciPy) as well as Fortran. Implementing EWMACD in Python took less than a week; implementing BFAST in Python took about 3 weeks; correctly implementing the recursive residuals based search for breakpoints took longest. Parallelization took 1.5 days. Fortran implementations took about 6 months, learning Fortran and mathematical libraries in the process. Parallelization in Fortran took about 3 weeks. The main difficulty was comprehending the use, limitations, and support of THREADPRIVATE vs. PRIVATE and then optimally declaring the corresponding variables.

5. Conclusions and future work

Python is an excellent language for developing code prototypes. Algorithms are implemented, analyzed, and tested in Python using minimal programmer time. Good scalability is achieved for NumPy-friendly expressions. Expressions that do not conform to NumPy constructs are slower and limit the scalability of the code. Experiments here indicate that Python’s cache utilization must be improved, especially with a large number of cores, for Python to offer performance comparable with Fortran. This paper makes a quantitative assessment (development time, performance, scalability) of Python codes vs. Fortran codes with respect to LULCC. Given the impending computation demands with these datasets, Fortran is indispensable. Development of Fortran codes takes longer but the codes are orders of magnitude faster and scale better. Strategically, therefore, code prototypes are developed in Python and then utilized to develop Fortran codes. The two algorithms used in this paper are intended to be parts of a polyalgorithm under development; for that, speeding up BFAST for a single pixel needs to be aggressively explored, lest BFAST be used only on a need basis (current run time is > 30 hours for a full scene with 198 time points).

References