DEVELOPING PARALLEL ALGORITHMS FOR SEASONAL METRICS EXTRACTION

Paul Marshall and Daniel Swets
Department of Computer Science
Augustana College
Sioux Falls, SD 57197

ABSTRACT

Temporal NDVI images can be used to show the change in vegetation on Earth over time. NDVI is derived from data produced by NOAA and NASA satellites, and scientists from NASA, NOAA, USGS and the CDC utilize these temporal images for land cover/land use change analysis, drought monitoring, fire management, weather prediction, flood analysis, and even the study of viral outbreaks, such as the Hantavirus outbreak in the southwestern United States. Seasonal metrics and vegetation characteristics must be calculated from these temporal images to enable the application of space-borne sensors to these studies of biological processes. Metrics such as the day when the season starts or ends, the total integrated NDVI, and others all need to be calculated for each pixel in the image. And because the global application of these studies requires a vast number of pixels, we developed a highly efficient software system to provide these seasonality metrics. The first phase of our project involved writing software to extract the metrics from each pixel. Due to the enormous size of the images we found that processing pixel by pixel required far too much running time to complete. Therefore, the second phase of our project implemented a threaded algorithm to enhance the speed and provide for overlapped file IO. The third phase of our project revolves around a clustering package to enable an unlimited number of networked computers to process different parts of the image. We have also written a Java interface to help control and monitor the processing of the images. Threading, overlapped file IO, and clustering will help NOAA, NASA, and USGS scientists calculate metrics closer to real time. This will enable them to utilize the satellite images more efficiently, and will enhance their ability to perform regional and global biological studies.

Keywords

Seasonal Metrics, Normalized Difference Vegetation Index, Threaded programming, Parallel Processing

INTRODUCTION

Space-borne sensors provide an extraordinary opportunity to study global seasonality changes. Vegetation indices, such as the Normalized Difference

Vegetation Index (NDVI), have been employed extensively for the past 25 years to characterize land cover, growth, and biomass production on a regional, national, and global scale. NDVI is computed from the near infrared and red-light regions of the spectrum[1]. An index such as NDVI enhances the vegetation signal, reduces background effects, provides some measure of data dimensionality reduction, all while providing a normalized metric with which to compare various land cover types and disparate regions[2,3,4]. We collect this NDVI data at regular time intervals for each pixel in our study region. These NDVI data can be noisy, due to various atmospheric perturbations, but reliable smoothing techniques can be employed to remove spurious fluctuations in the signal[5]. Seasonal metrics and vegetation characteristics must be calculated from these temporal images to enable the application of space-borne sensors to studies of biological processes. Metrics such as the day when the season starts or ends, the total integrated NDVI, and others all need to be calculated for each pixel in the image. Existing methods[6,7] for the extraction of the seasonality metrics from satellite data were written in an interpreted image processing language, and were therefore too slow for timely analysis or for real-time processing. Furthermore, the values computed by these methods had not been verified, and the accuracy of some of the values was suspect. We have designed a highly efficient software package to calculate seasonality metrics from these temporal NDVI images.

METHODS

We began our project by gaining a thorough understanding of the metrics that our program would need to extract from the images: Start of Season, End of Season, Peak of Season, Amplitude, Season Duration, Rate of Greenup, Rate of Senescence, and the Integrated NDVI. Figure 1 illustrates an NDVI curve

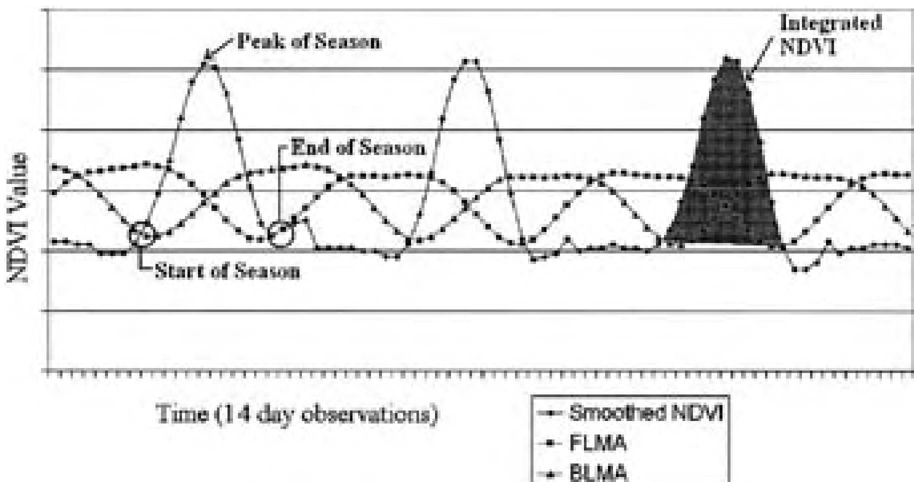

Figure 1. Sample NDVI data curve. We have added a backward-looking moving average (BLMA) and a forward-looking moving average (FLMA).

and shows a sample of some of the metrics we compute.

The Start of Season is found at the point where the backward-looking moving average crosses the NDVI data curve. The backward looking moving average is found by looking backward on the data curve and taking the average of *n* data points, where *n* is the user-specified number of data points to include in the average calculation. Empirical studies have found n=15 to provide an accurate start-of-season for well-behaved unimodal and bimodal season curves. The End of Season is found where the forward-looking moving average crosses with the NDVI data curve. The forward-looking moving average is calculated by looking ahead *m* points, where *m* is the number of points to include in the average calculation. We have found empirically that m=18 provides a good calculation for the end-of-season. The Peak of Season is the point with maximum NDVI value between the start-of-season and end-of-season. Season Duration is the length of time between the start- and end-of-seasons. The Rate of Greenup is the slope between the start-of-season and the peak of the season. Rate of senescence is the slope between the peak of the season and the end-of-season. The Integrated NDVI is the area under the NDVI data curve between the start-of-season and the end-of-season (see the shaded area in Figure 1). The amplitude is the difference between the peak NDVI value and the start of season NDVI value.

The NDVI images produced by the satellite are in row-raster format; each pixel in the image is appended to the previous pixel and subsequent images are appended to the previous image. This format creates a series of images that appear to be one long string of bits, as shown in Figure 2.

Image 1 from time t	Image 2 from time t+1	Image 3 from time t+2
01010001000101111101101011	11000100010010100101111011	10001010101101101001101 1

Figure 2. Row-raster format for storing NDVI temporal images. We stack images taken from different time periods for storage.

Per-Pixel System

We developed a proof-of-concept system to extract all these metrics on a per-pixel basis. Our read function reads a single pixel through all of the temporal images. After we read pixel one from image one the program seeks past the rest of the image to get to the first pixel in image two; this process continues until it gets through all of the images. A sample of the code to implement this action is given in Figure 3.

After we read in one pixel through all of the bands we scale the pixel by 100 in order to limit floating-point operations, which perform slower than simple integer operations. This is done by the line: pixel_data[i] = pixel[i] * 100;

Once a pixel was read into pixel_data we then used this to calculate the metrics on that pixel and store the results in a separate array. Our initial version first calculated the Start of Season and End of Season on the pixel because those values were needed before we could calculate any of the other metrics. The first

```
//DESC: A function to read one pixel in an NDVI image into an array and scale the NDVI
      value by a factor of 100.
//PRE:- myFile is a pointer to the file that we are reading from
      - num_bands is the number of bands or images in myFile (i.e. a file with an image
      taken once per week for a year would have 52 bands.
      - pixel_num is the pixel that we are going to read, starting at 0.
      - image_size is the total number of pixels in one image (there will be the same number
      of pixels in all of the subsequent images.
//POST: the scaled pixel is placed in pixel_data which is a double array of num_bands
      items.
void read_func(FILE * myFile, int num_bands, int pixel_num, int image_size, double *
      pixel_data) {
char * pixel;
pixel = new char[num_bands];
for(int i=0; i<num_bands; i++)
{
      fseek(myFile, (pixel_num+(i*image_size)), SEEK_SET) ;
      fread(&(pixel[i]), 1, 1, myFile);
}
for(i=0; i<num_bands; i++)
{
            pixel_data[i] = pixel[i] * 100;
}
delete[ ] pixel;
}
```

Figure 3. C++ listing of a per-pixel read algorithm.

thing we did to calculate the Start of Season and End of Season was to calculate the forward looking moving average (FLMA) and the backward looking moving average (BLMA), as shown in Figure 4.

The forward looking moving average is calculated in a similar manner, however, instead of looking backward on the NDVI data curve the program looks forward.

After calculating the forward and backward looking moving averages the program locates where the BLMA crosses with the NDVI curve, which is the Start of Season. We created a check in our program to make sure that the season was long enough in order to avoid the BLMA curve crossing with the NDVI curve too soon for there to actually be a season. Once we located the Start of Season we then proceeded to locate the End of Season by finding where the FLMA crossed with the NDVI data curve. We provide a similar check for the End of Season to make sure the FLMA cross with the NDVI curve is legitimate.

Once we have located the Start of Season and End of Season values for the pixel we then calculated the rest of the metrics on that pixel. Most of the other metrics are simple calculations once the Start of Season and End of Season have been identified. To calculate the Integrated NDVI we calculate the area between the Start of Season and End of Season by breaking up each observation into a triangle and rectangle as shown in Figure 5a, computing the area and then add-

```
//Backward looking moving average
int posi=0;
int position=0;
int sum=0;
while(posi<num_bands)
{
	while(position<num_BLMA_points)
	{
		sum+=my_pixel_data[position+posi];
		position++;
	}
	myBLMA[posi] = sum / num_BLMA_points;
	position = 0;
	sum = 0;
	posi++;
}
```

Figure 4. Backward-looking moving average calculation.

ing up all of the values between each of the observations. After this has been completed the area below the line connecting the start-of-season point with the end-of-season point must be subtracted from the initial total, as shown in Figure 5b.

The values that the user wants to be calculated are given to the program in an arguments text file. The user specifies the number of seasons expected in the data series, and each of the metrics being computed is stored in an array whose length is that number of seasons.

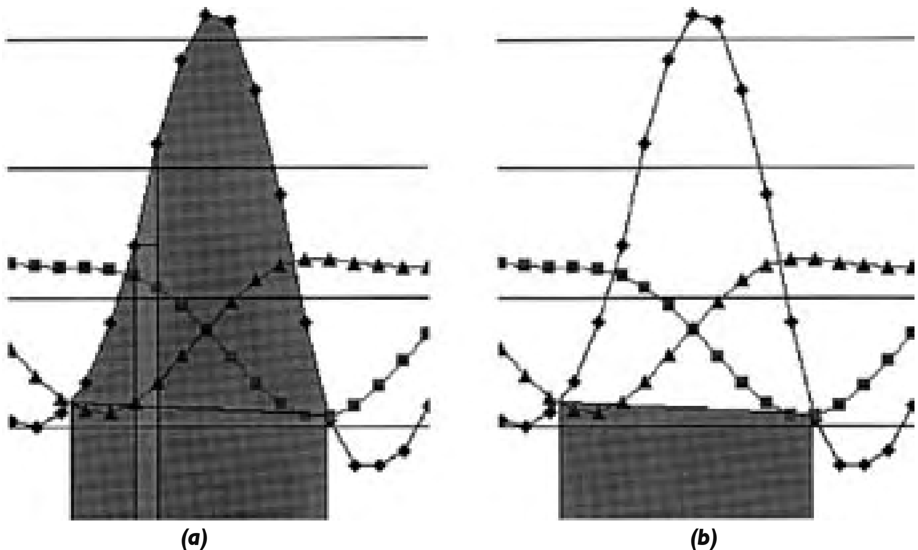

Figure 5. Integrated NDVI calculation. (a) Triangle and rectangle portion for area calculation. (b) Subtracted portion below the line connecting the start-of-season point with the end-of-season point.

Addressing Input and Output

This initial design completed much faster than the previous versions; however, we realized that the file access (reading/writing) was a major bottleneck for our program. Our next versions were an attempt to utilize features of modern computers, such as threads and I/O Completion Ports to obtain maximum efficiency.

A run-time analysis showed that most of the real time elapsed was spent waiting for file reads and file writes to complete. To address this, we added a mechanism for reading a large group of pixels at a time. The read function read the one-byte data into a char array and then waited until the program needed a particular pixel to scale it by a 100 into a double, as shown in Figure 6.

```
int read_sector_size=100000;
int low=0;
int high=0;
int index=0;
void read_func(FILE * myFile, int num_bands, int pixel_num, int image_size, double *
pixel_data)
{
	static unsigned char ** sectorarray = NULL;
	int i=0;
	if(sectorarray == NULL)
	{
		sectorarray = (unsigned char **)(malloc(num_bands * sizeof(*sectorarray)));
		for(i=0; i<num_bands; i++)
			sectorarray[i] = (unsigned char *)(malloc(read_sector_size *
				sizeof(**sectorarray)));
	}
	if(index == read_sector_size)
	{
		for(i=0; i<num_bands; i++)
		{
			fseek(myFile, (pixel_num+(i*image_size)), SEEK_SET);
			fread(sectorarray[i], 1, read_sector_size, myFile);
		}
		low=high;
		high+=read_sector_size;
		index=0;
	}
	for(i=0; i<num_bands; i++)
	{
		pixel_data[i] = sectorarray[i][index] * 100;
	}
	index++;
}
```

Figure 6. Reading a multiple of a disk sector rather than a single pixel at a time.

This version produced much better results than our initial version of the program; however, we were still unsatisfied with the speed.

Threaded Approach

In threaded programming, a single program is capable of executing in several places simultaneously. This incurs some additional overhead, such as keeping track of which thread is working on which portion of the data, but it allows for processing to be completed where the system would otherwise be waiting for external events to occur. For our application, we developed a suite of threads, each of which responsible for reading, processing, and writing. The only trouble with implementing threads was keeping track of which thread had which group of pixels so each thread knew where to write its results. This was solved by a few global structures to maintain an identifier for which group of pixels each thread is manipulating. This version made a huge improvement on a dual-processor machine, but did not show a marked improvement on a single processor machine. In fact we actually saw a slightly slower time on the single processor machine.

I/O Completion Port

Finally, we broke the various functions of our program into separate threads and utilized I/O Completion Ports to handle the signaling between the threads. We created a single *read thread* to handle all of the reading; three *processing threads* to handle all of the processing and a single *writing thread* to write the results out to the disk. We also created pixel blocks to hold the data in between the stages of reading, processing, and writing, as shown in Figure 7.

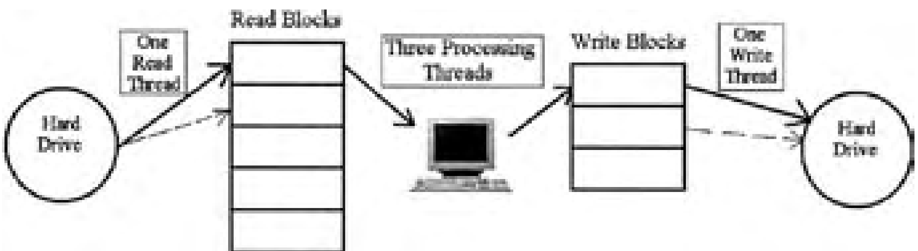

Figure 7. Threaded programming with I/O completion ports and pixel blocks to interface between the functional threads.

The *read thread* loops and fills the read blocks with groups of pixels. When the entire set of read blocks have been filled, the read thread waits for a signal that another read block is empty and ready to be filled. The structure for the read thread is shown in Figure 8.

```
#define NUM_READ_BLOCKS 5
HANDLE completionPortIO;
    completionPortIO = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL,
    0, NUM_PROCESSING_THREADS);
HANDLE WorkItemFinishedEvent;
WorkItemFinishedEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
volatile bool * Block States;
    BlockStates=new bool[NUM_READ_BLOCKS];
for(int i=0; i<NUM_READ_BLOCKS; i++)
    BlockStates[i]=false;
while(CURRENT_PIXEL<image_size)
{
    BOOL bReadFromFile=false;
    for(int i=0; i<NUM_READ_BLOCKS; i++)
    {
        if(BlockStates[i]==false)
        {
            BlockStates[i]=true;
            //call read_func();
            //increment CURRENT_PIXEL
            bReadFromFile=true;
                PostQueuedCompletionStatus(completionPortIO,
                SIZE_OF_BLOCK, i, NULL);
        }
    }
    if(!bReadFromFile)
    {
        WaitForSingleObject(WorkItemFinishedEvent, INFINITE);
    }
}
```

Figure 8. Read thread structure on a win32 system.

The *processing thread* takes a block of pixels from the read blocks and processes them. These threads have to wait for a read block that has been successfully filled, and the processing thread listens for the read thread to call PostQueuedCompletionStatus.

We use a timeout of five seconds to indicate that all of the blocks have been processed and there are no more pixels to be processed. As soon as a processing thread fills its array of results it places the results in a write block and notifies the write thread that it has data to write out. The write thread then locates the proper block in the file and writes it out.

RESULTS

We ran the metrics program on two separate machines. We used an image of the conterminous United States for all of the tests. The image was 2889 rows by

4587 columns with 52 bands; it was 672,946 KB. For testing purposes, machine one was an AMD 2500 with 1 GB of RAM, and machine two was a dual Celeron, each processor running at 400 MHz, with 196 MB of RAM. The initial per-pixel version of the metrics ran on machine one in one hour and 18 minutes. While it performed much faster than the IDL metrics, which could calculate the metrics in one hour and 38 minutes on a machine with a 3.2 GHz processor and 2 GB of RAM, we were not impressed with the running time. The per-pixel version completed on the dual Celeron in 21 hours and 49 minutes, which was also unimpressive. However, after we had addressed some of the reading and writing issues with our second version, the metrics completed much faster. With blocked I/O, machine one completed in eight minutes and 58 seconds and machine two completed in one hour and six minutes. The threaded version completed in 23 minutes on machine two, and it completed in nine minutes and 49 seconds on machine one. Our final version utilizing threads and IO Completion Ports saw a drop in time on both machines. Machine one completed in six minutes and 55 seconds; machine two completed in 19 minutes and 56 seconds.

DISCUSSION

The main reason that there was such a huge improvement from the first version of the application (per-pixel) to the second was due to the fact that the program had much less disk access. We saw another large performance increase on our dual Celeron in our third version when we added threads; adding threads allowed both processors to be utilized. The reason for the drop in performance on the AMD is because of the context switching that the processor has to do when threads are used. Unless separate threads are used appropriately to do varying tasks on a single processor machine, threads do not necessarily provide an advantage. Multiple threads all executing the same code can slow the machine down, as demonstrated here. Multiprocessor and hyper-threaded machines can actually execute multiple threads simultaneously so threads are an advantage for them, even if the threads are executing the exact same code. Threads did provide an advantage on the single processor machine (AMD) when we separated the reading and writing functions from the processing code and employed an efficient communication system between the threads.

Future versions of this software could implement parallel prefix[8] calculations to take advantage of massively parallel architectures like the ncube at the USGS National Center for Earth Resource Observation System (EROS) in Sioux Falls, South Dakota. Parallel prefix calculations can perform operations such as summation, locating a maximum, and locating a minimum extremely efficiently; such operations can be completed in O(logn) time on n items with n processors. This would allow the most time intensive calculations such as the Start of Season, End of Season, and Integrated NDVI to be derived in real time.

ACKNOWLEDGEMENTS

This work was supported in part by NASA Grant number NGT5-40042.

LITERATURE CITED

[1] Tucker, C.J. 1979. Red and photographic infrared linear combinations for monitoring vegetation. Remote Sensing of the Environment, v. 8, p.127-150.

[2] Curran, P.J. (1981). Multispectral remote sensing for estimating biomass and productivity. In: Plants and the Daylight Spectrum (H. Smith, editor), Academic Press, London.

[3] Malingreau, J.P. (1989). The vegetation index and the study of vegetation dynamics. In: Application of Remote Sensing to Agrometeorology, (F. Toselli, editor), ECSC, Brussels and Luxembourg, pp. 285-303.

[4] Goward, S.N. (1989). Satellite bioclimatology. Journal of Climate, 2:710-720.

[5] Daniel L. Swets, Shaun E. Marko, James Rowland, Bradley C. Reed, "Statistical Methods for NDVI Smoothing," in Proceedings, American Society for Photogramity and Remote Sensing, May, 1999.

[6] Reed, B.C., J.F. Brown, and D. VanderZee (1994). "Measuring phenological variability from satellite imagery." Journal of Vegetation Science, 5:703-714.

[7] Defries, R., M. Hansen, and J. Townshend (1995). Global discrimination of land cover types from metrics derived from AVHRR pathfinder data. Remote Sensing of the Environment, 52:209-222.

[8] Miller, Russ, et. al, Algorithms Sequential & Parallel, Prentice Hall, 2000.