Concurrent and Parallel Systems - Assignment Report

1 Description of Algorithm Design for Parallelism / Concurrency

Problems Begin Solved

In this assignment two algorithms were implemented to solve problems from two separate domains. One algorithm performs simple image processing where an image kernel or convolution matrix is applied to pixels in an image to produce a new filtered image with some effect (blur, sharpen, edges). The second algorithm is a very basic implementation of a “grep” program that counts the number of occurrences of a target word in all the text files in a given directory and its subdirectories recursively, it also notes the location of the word in each file with a line number and column number.

Core Algorithm that Solves Problem

At their core each implementation makes use of a key algorithm or piece of code that does the low level work to solve the problem.

The image processing implementation uses a function called **applyKernelPixel**, it works by taking the kernel and calculates what a pixel in the source image, specified by x & y coordinates, will look like with the kernel applied to it. This is done by taking the values of the pixels surrounding the target pixel, multiplying them by their corresponding values in the kernel, then summing the results up to get a final value for the output pixel. This value is then put into the output image at the specified x & y location.

The simple grep implementation makes use of a function called **searchBytes.** The function takes a byte buffer and a target string, it works by iterating over the byte buffer and counting the number of consecutive bytes that match the target string, when this reaches the length of the target string a match is recorded and the location of the match in the file is noted. The function returns the total number of matches and all the locations in the file where a match was recorded.

How the Sequential Version Works

The sequential version of the simple grep implementation works by generating a list of all the files to open in the given directory and all directories below it, this list of files is then iterated over in a sequential fashion and searched using the **searchBytes** function. Each file is opened and the contents loaded into a byte buffer which is passed, with the target string, to the **searchBytes** function.

The sequential version of the image processing implementation works by loading a source image from a file into a buffer and then iterating over every pixel in the buffer, using the **applyKernelPixel** function to calculate each pixel’s value with the kernel applied to it. The x

& y coordinates of each pixel is passed into the function sequentially and the function then places the result into an output buffer to create the output image.

How the Parallel Version Works

Being written in Go both implementations make use of its concurrency features, by design these implementations are “concurrent” as multiple threads can make progress through the workload they are given even though they might not be running at the same time (multitasking via time-slicing) due to only being able to use one CPU. Parallelism is only achieved at run-time, when multiple threads can execute simultaneously, this can only be done with multiple CPUs. In Go the max amount of CPUs that the program can use can be specified.

Simple Grep

The concurrent / parallel version of the simple grep implementation works by generating a list of all the files to open in the given directory and all directories below it, a “worker” Go routine is then launched for each file in the list, the routines then wait to be given “jobs” on the job channel.

A job is the name of the file a given routine will search, before beginning searching the routine has to acquire the okay to begin from a channel that counts the amount of files that are open, this prevents too many files being opened. When it acquires the okay the counter goes up on the channel (which has a max, it will block routines from continuing - and thereby opening files, until the counter comes back down).

When the routine is finished searching it puts the results from the search into channels which are then picked up by “collector” routines, after this the routine releases the okay it acquired earlier from the “opened file counter” channel so other routines can continue.

The search is complete when all the worker routines finish and all the collector routine have put the final result together.

Image Processing

The concurrent / parallel version of the image processing implementation works by loading a source image from a file into a buffer and then iterating over each row in the source image, each row is then sent to a Go routine to have the kernel applied to it. The Go routine iterates over the row of pixels it is given using the **applyKernelPixel** function on each pixel, since each pixel has its own destination in the output image buffer it can be written to by each Go routine without the use of channels to share access to the buffer.

The algorithm is complete when all the Goroutines have finished executing and the output image is then saved to disk.

2 Representative Parallel / Concurrent Source Code

Simple Grep

Main function **searchFoldersPara** that creates all the channels and kicks off all the worker, jobMaker and collector routines to perform the searching.

```
// search the folders provided in the arguments to the program - search is done in parallel
func searchFoldersPara(verboseOutput, fileCountMap *string, fileCount, charCount, fullCount *int64) {

       *fileCount = int64(len(fileList))
       var wg sync.WaitGroup

       fileListLen := int64(len(fileList))
       openedFiles := make(semaphore, cpuCount)
       fileJobsChan := make(chan string, cpuCount)
       verboseOutputChan := make(chan string, cpuCount)
       fileCountMapChan := make(chan string, cpuCount)
       occurrenceCountChan := make(chan int64, cpuCount)
       charCountChan := make(chan int64, cpuCount)

       // kick off a worker routine for each file in fileList, each waits  wait to be given jobs
       for i := range fileList {
               // increment the wait group counter for each routine launched
               wg.Add(1)
               go worker(searchStrFlag, fileJobsChan, verboseOutputChan, fileCountMapChan,
                       charCountChan, occurrenceCountChan, &wg, i, openedFiles)
       }

       // kick off routine to create jobs
       wg.Add(1)
       go jobMaker(fileJobsChan, fileList, &wg)
       // collect the results created by the worker routines
       wg.Add(4)
       go occurrenceCountCollector(fullCount, occurrenceCountChan, fileListLen, &wg)
       go fileCountMapCollector(fileCountMap, fileCountMapChan, fileListLen, &wg)
       go verboseOutputCollector(verboseOutput, verboseOutputChan, fileListLen, &wg)
       go charCountCollector(charCount, charCountChan, fileListLen, &wg)

       // wait until all the routines in the wait group complete before continuing
       wg.Wait()
       // close all the collection channels once
       close(verboseOutputChan)
       close(fileCountMapChan)
       close(occurrenceCountChan)
       close(charCountChan)
}
```

jobMaker routine which is responsible for giving each worker routine a file to search from the file list

```
// routines that "makes" jobs (filenames) and puts them in a channel for workers to receive
func jobMaker(fileJobsChan chan<- string, fileList []string, wg *sync.WaitGroup) {
       defer wg.Done()

       for _, fileName := range fileList {
               fileJobsChan <- fileName
       }
}
```

worker routine which is responsible for performing the searching of the file it is given and the putting the results into a channel to be picked up by collector routine

```go
// routine that performs the actual searching task
func worker(targetStr string, fileJob <-chan string, resultsVerboseChan,
resultsFileCountMapChan chan<- string, charCountChan, totalCountChan chan<- int64, wg *sync.WaitGroup,
id int, openedFilesSem semaphore) {

	defer wg.Done()
	originFileName := <-fileJob
	// acquire the okay for opening files
	sem := empty{}
	openedFilesSem <- sem
	// open and read from file
	fileData, err := ioutil.ReadFile(originFileName)

	if err == nil {
		// if name was qualified chop it down to the base / showten if need be
		fileName := filepath.Base(originFileName)
		if len(fileName) > 25 {
			fileName = fileName[:25] + "..."
		}
		// search the given file for the target string then put results into channels
		if true == progressFlag {
			fmt.Print(fmt.Sprintf("Searching file: %s \n", fileName))
		}
		output, numFound, numChars := searchBytes(targetStr, fileData, fileName)
		// put results from the search back into corresponding channels
		totalCountChan <- numFound
		charCountChan <- numChars
		resultsVerboseChan <- output
		resultsFileCountMapChan <- fmt.Sprintf("%s : %d occurrences\n", fileName, numFound)
	} else {
		check(err)
	}
	// release the okay for opening files
	<-openedFilesSem
}
```

The four **collector** routines that collect information from the worker routines on various channels.

```go
// routine to collect the number of matches from occurrenceCountChan and put it into fullCount
func occurrenceCountCollector(fullCount *int64, occurrenceCountChan <-chan int64, items int64, wg
*sync.WaitGroup) {
	defer wg.Done()
	var sumOccurences int64

	for i := int64(0); i < items; i++ {
		sumOccurences += <-occurrenceCountChan
	}

	*fullCount = sumOccurences
}
```

```
// routine to collect the verbose output from the verboseOutputChan and put it into the string verboseOutput
func verboseOutputCollector(verboseOutput *string, verboseOutputChan <-chan string, items int64, wg
*sync.WaitGroup) {
	defer wg.Done()
	var verboseOutputBuffer bytes.Buffer

	for i := int64(0); i < items; i++ {
		verboseOutputBuffer.WriteString(<-verboseOutputChan)
	}

	*verboseOutput = verboseOutputBuffer.String()
}




// routine to collect the file names with their occurrence counts from fileCountMapChan and put it into
fileCountMap
func fileCountMapCollector(fileCountMap *string, fileCountMapChan <-chan string, items int64, wg
*sync.WaitGroup) {
	defer wg.Done()
	var fileCountMapBuffer bytes.Buffer

	for i := int64(0); i < items; i++ {
		fileCountMapBuffer.WriteString(<-fileCountMapChan)
	}

	*fileCountMap = fileCountMapBuffer.String()
}




// routine to collect the number of characters from charCountChan and put it into charCount
func charCountCollector(charCount *int64, charCountChan <-chan int64, items int64, wg *sync.WaitGroup) {
	defer wg.Done()
	var sumChars int64

	for i := int64(0); i < items; i++ {
		sumChars += <-charCountChan
	}

	*charCount = sumChars
}
```

Image Processing

Section showing how the image is processed concurrently using anonymous Goroutines to work on each row of the source image

```
// parallel operation ------------------------------------------------------
	var wg sync.WaitGroup // wait group to synchronize routines

	fmt.Print("Begin parallel\n")
	startTimePara := time.Now()

	// iterate through each row of the image stored in the NRGBA struct
	for rowY := beginY; rowY < outHeight; rowY++ {
		// increment the wait group counter for each routine launched
		wg.Add(1)

		// use anonymous go routine to sends each row to be processed concurrently
		go func(row, begin, end int, src *image.NRGBA, dest *image.NRGBA, kernel []float64,
			 wg *sync.WaitGroup) {

			defer wg.Done()

			// loop through each pixel in the row given to the routine and apply the kernel to it
			for pixel := begin; pixel < end; pixel++ {
				applyKernelPixel(pixel, row, src, dest, kernel)
			}

		}(rowY, beginX, outWidth, srcImgNRGB, outImagePara, selectedFilter, &wg)
	}
	// wait until all routines in the wait group finishes before continuing
	wg.Wait()

	// get operation metrics
	elaspedTimePara := time.Since(startTimePara)
	fmt.Print("End parallel\n")
	elaspedInSecondsPara := elaspedTimePara.Seconds()
	pixelsPerSecondPara := float64(imagePixelCount) / elaspedInSecondsPara

	outputName = fmt.Sprintf("%s_%s_output_para.jpg", nameNoExtension, selectedFilterStr)
	err = imaging.Save(outImagePara, outputName)
	if err != nil {
		panic(err)
	}
```

3 Justification and Description of Performance Metrics

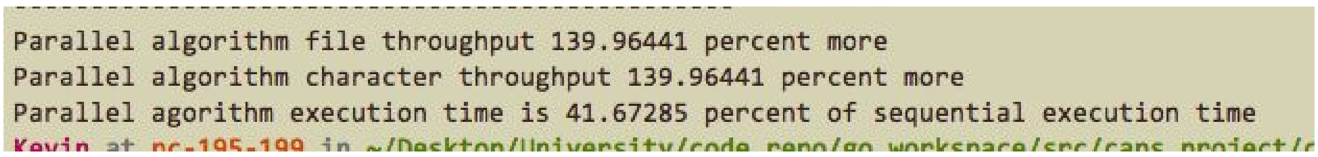
Both the Simple Grep and Image processing implementations make use of domain specific metrics to show the performance of each algorithm. Simple Grep outputs the number of characters scanned per second as well as the amount of files processed per second, the Image Processing program outputs the number of pixels operated on per second. Both implementations also output how long it took the algorithm to complete.

These performance metrics were chosen as they are specific to the domain that each algorithm is tackling, this makes it easily relatable to the problem and puts the amount of work being done into scale. In particular characters per second and pixel per second were chosen as they are the smallest unit of work the problems can be broken down into.

In addition using these metrics makes it much easier to compare the performance of the sequential algorithm compared to the parallel algorithm as they specify exactly how much *work* is being done, or how much more work one approach is doing compared to the other. This is more informative than comparing execution time as this only tells us how long the algorithms took to complete, there might not be a huge amount of difference in execution time but the change in *throughput* is typically affected significantly.

Both programs also output information that compares the throughput in the form of percentages, the throughput increase of the faster algorithm is output , this is useful as it directly compares the throughput of both the sequential and parallel/concurrent method. Also the execution time of the faster algorithm is output as a percentage of the execution time of the slower algorithm, essentially showing "how much" of the sequential algorithms execution time the parallel implementation took to complete, this is an intuitive way of comparing the execution time.

Percentages were included as they are a simplified way of comparing the differences between the performance metrics of each approach and are easier to put into perspective compared to "eyeballing" raw numbers, and especially when the numbers get very large.



```
----------------------------------------------
Parallel algorithm file throughput 139.96441 percent more
Parallel algorithm character throughput 139.96441 percent more
Parallel agorithm execution time is 41.67285 percent of sequential execution time
Kevin at pc-195-199 in ~/Desktop/University/code_repo/go_workspace/src/cans_project/c
```

Example of percentages being used to compare sequential and concurrent / parallel implementations

4 Critique, Evaluation and Suggestions For Future Improvements

In looking at the general trend of the performance graphs presented at the end of the report, it is clear that the concurrent / parallel implementations offer a great deal of performance increase compared to their sequential counterparts whose performance stays consistent regardless of the number of CPU's available. In all of the graphs, for both simple grep and image processing, the general trend is that the execution time of the parallel implementation decreases as the number of CPUs used increases, and the throughput of the parallel implementation increases as the number of CPUs used increases. This decline in execution time and increase in throughput appears to be roughly logarithmic.

It is also clear from these graphs that the performance increase begins to taper off fairly quickly even though a maximum of only 4 CPUs were used. This means that the performance increase from using more CPUs has some sort of upper limit.

The concurrent / parallel implementation for both Simple Grep and Image processing could be described at best as naive solutions that demonstrate the core concepts behind concurrency/ parallelism and show that serious performance gains can be had even with simplistic solutions. However there are a number of problems that are present in each implementation that can be fixed to improve the general performance and scalability of the solutions.

The first issue that is actually present in both implementations is the lack of any attempt to determine how much work has been given to the algorithm before acting. Doing this at the beginning would be useful, as an estimation of the amount of work that needs to be performed could be used to dynamically decide how the algorithms behave , particularly with very small and very large workloads.

Most of the issues highlighted are to do with the scalability of the solutions.

Simple Grep

The first major issue present in the Simple Grep program is how the file directory is scanned for files that will be searched, currently the algorithm waits for the file list to be made before beginning to search each file. An improvement to this would be to make the creation of this file list run in a Go routine of its own and send the names of files it has found to a large buffered channel, another "job maker" Go routine can then wait on this channel and immediately begin sending jobs to worker channels that do the searching whilst the file directory is still being scanned.

In addition to this another issue is the way collection channels are used, currently each Go routine that searches a file produces 4 pieces of data and sends this data to be collected on 4 separate channels which have 4 other Goroutines collecting the data from them, this is an overly complicated design. An improvement to this would be to create a struct to store all the data the worker routines create on search completion and use a single channel to send and collect this data on.

One last key issue is the way in which search results are built up, one of the pieces of data that is returned from worker threads, which do the searching, is the positions of where matches are found in each file as a string. This can get very large when this is put together as a final result from all the files searched if the directory being searched contains many files with many matches, and since this is done in memory it uses up vast amounts of RAM - some operating systems compress a programs RAM usage when it exceeds a certain amount to conserve space which can then slow down the performance of the algorithm. An improvement to this would be to stream this output into a file on disk while the searching is happening after memory usage goes over a certain threshold, a Go routine could be dedicated to doing this task.

Image Processing

The first key issue present in the image processing program is how inefficient it is with using memory. The entire image is loaded into one buffer before operating on it and makes use of an output buffer of at least the same size, this is unfeasible for very large images or computers with small amounts of memory. A solution to this could be to operate on very large images in portions, loading a portion, then operating on it before writing it to an output file, this would only be necessary for very large files or on machines with small amounts of memory. A dynamic detection system could be used to determine when to do this.

One other closely related issue is the way concurrency is applied to solve the problem, a Go routine is launched for each row in the image, again for very large images this could be unfeasible and there is no need to launch all these Goroutines at once when the amount that can actually perform work is limited by the number of CPUs on the machine. An improvement to this would be to take the portion of the very large image that is loaded and dynamically chop it up into evenly sized blocks which can be operated on concurrently, a load balancing system could then be used that determines the amount of Goroutines to launch based on the number of CPUs in the machine, the load balancer can then launch new Goroutines when a CPU finishes working on a previous block.

5 Result From Execution Runs

To get an idea of how each concurrent / parallel implementation performs on average compared to the sequential implementation the programs were run a number of times, with different configurations. A configuration is the workload size and the amount of CPU's that the program can use (r*anging from 1 - 4 CPUs*) , each configuration was run 5 times to get an average. The results are presented in the graphs following this page.

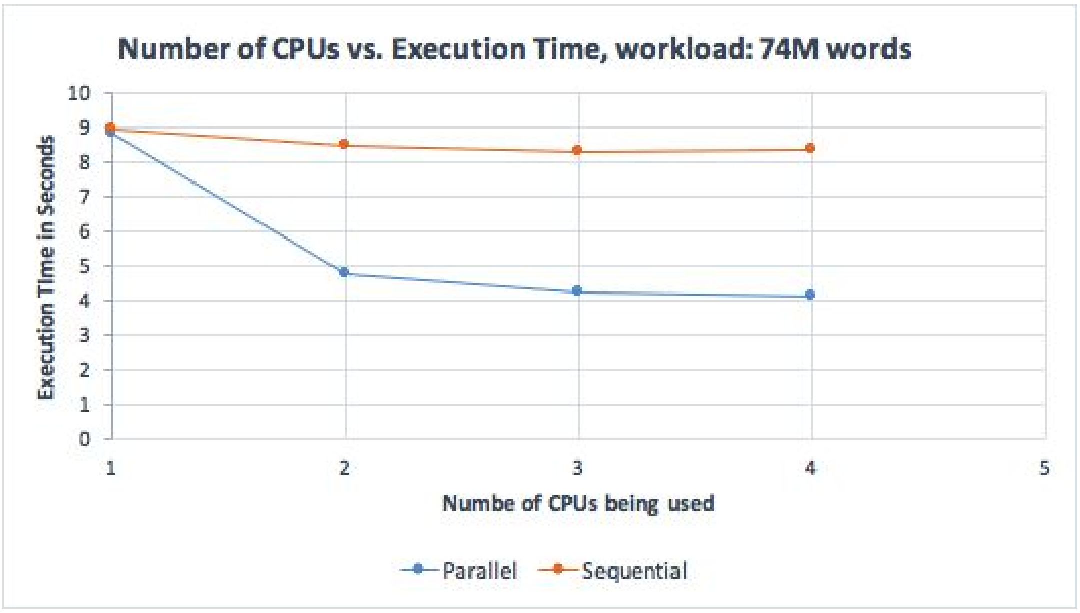
The tables below show the workload sizes used for the Simple Grep and Image processing programs.

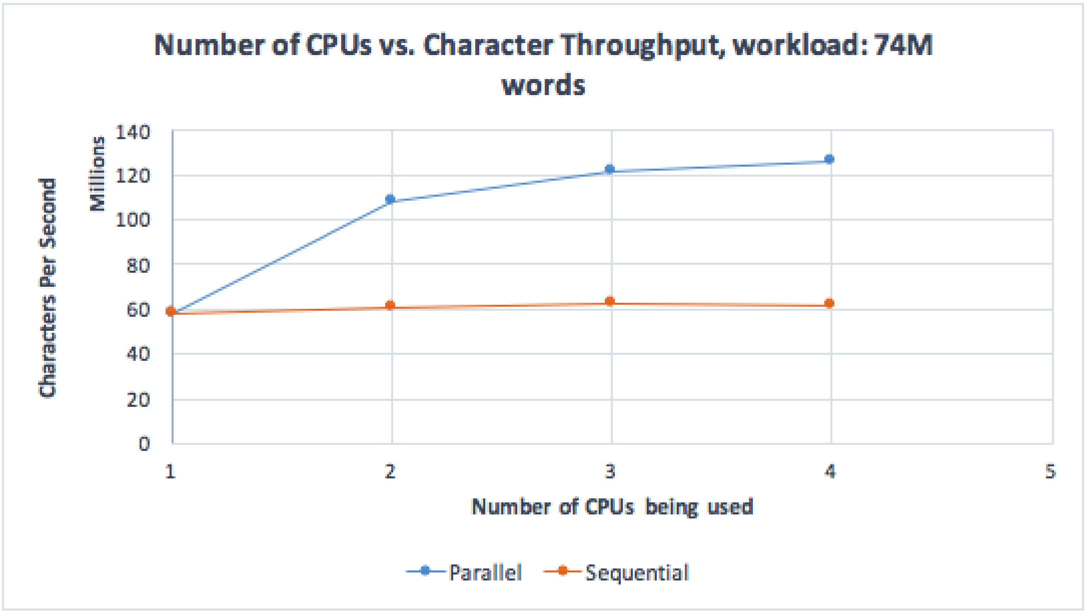
Image Processing		
Configuration	**Total Pixels**	**Size on Disk**
small	100,000	24 KB
medium	1,000,000	152 KB
large	10,036,224	3.7 MB
very large	100,800,320	19.4 MB

Simple Grep Search			
Configuration	**Total Words**	**Total Files**	**Size on Disk**
small	74,647,954	19,218	515.6 MB
medium	149,295,904	38,435	1.03 GB
large	298,591,465	76,869	2.06 GB

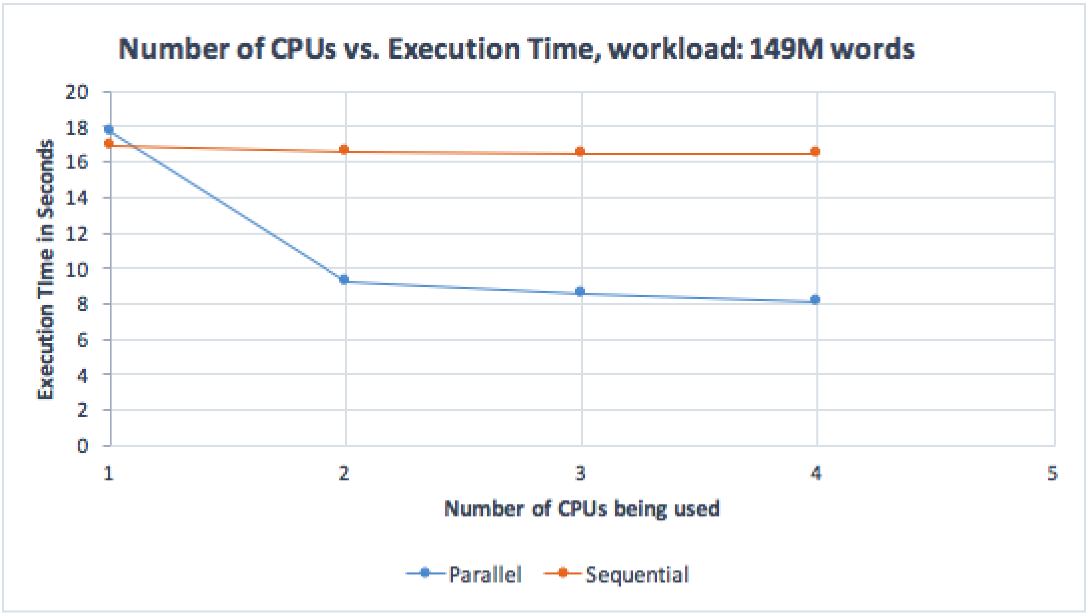
Simple Grep Performance Graphs

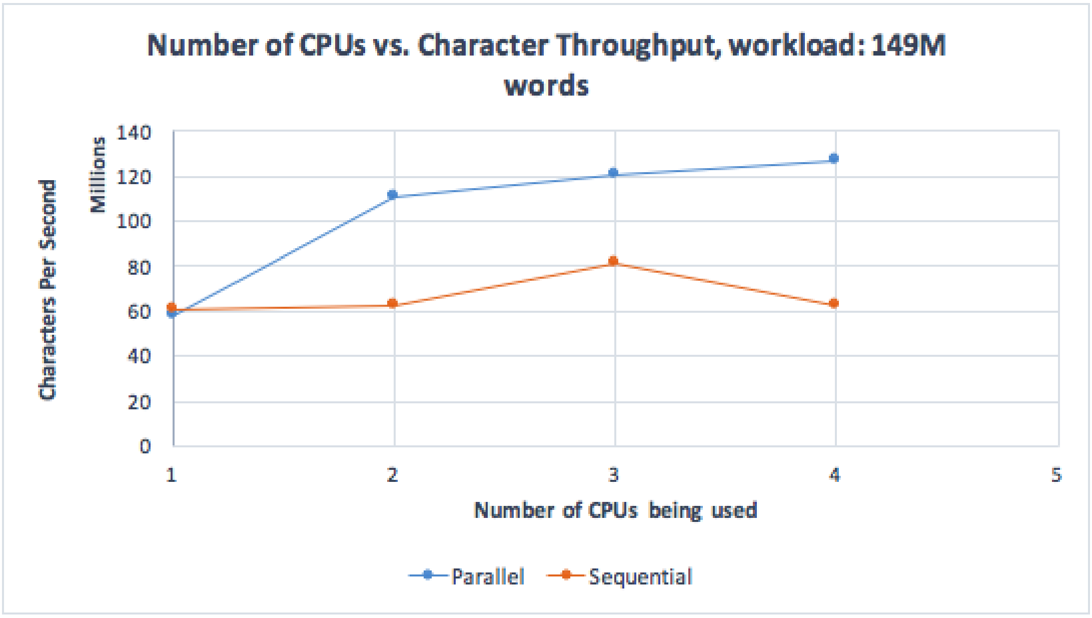
Performance with small data set, plain text corpus with 74,647,954 words



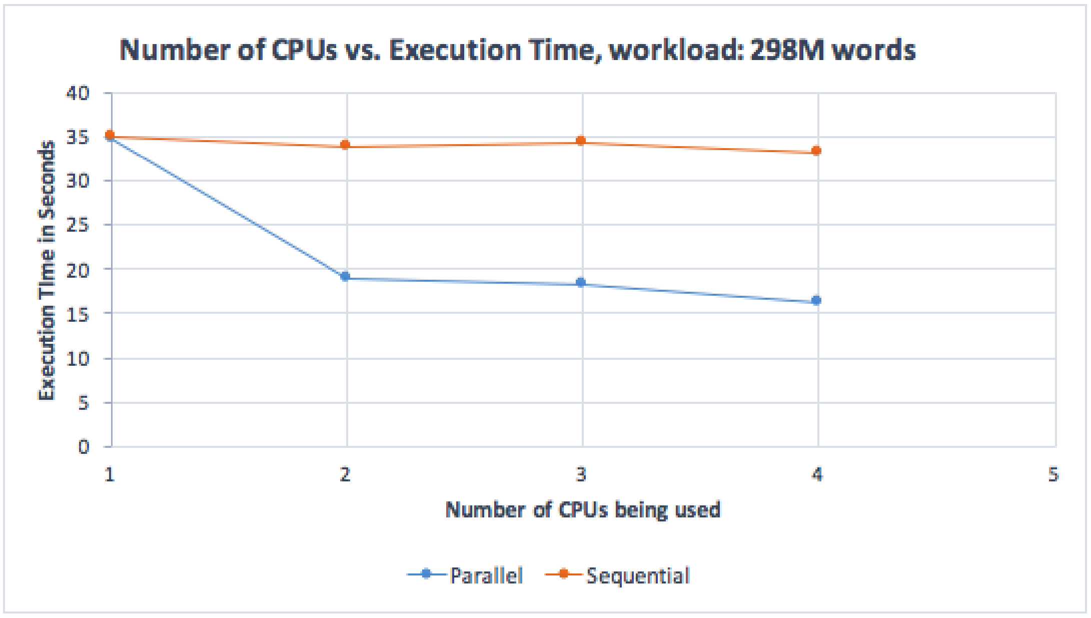


Performance with medium data set, plain text corpus with 149,295,904 words





Performance with large data set, plain text corpus with 298,591,465 words



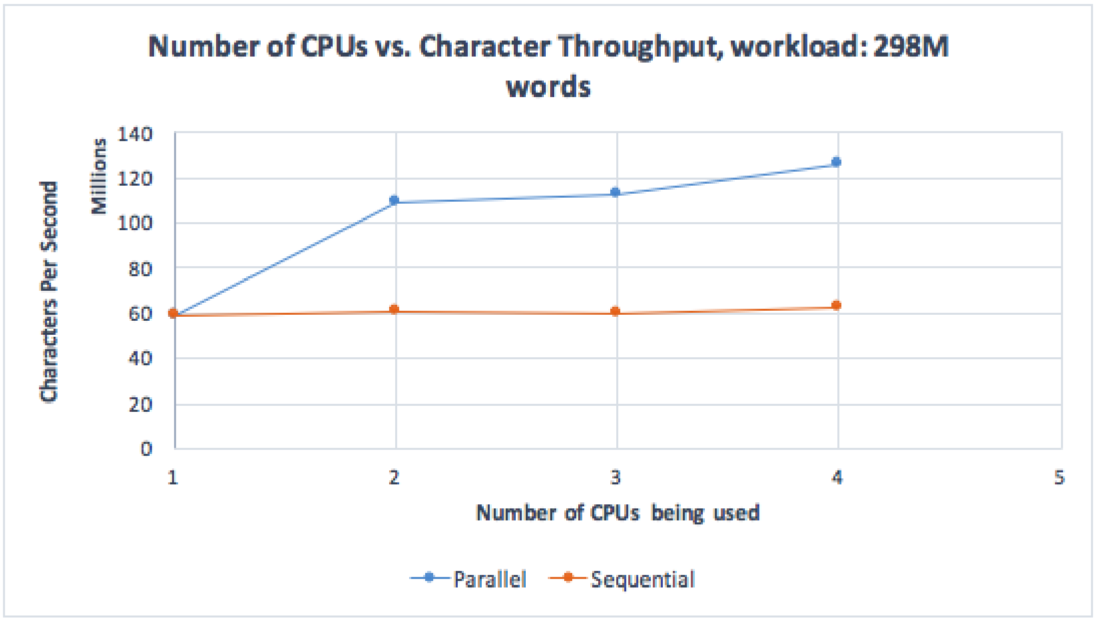
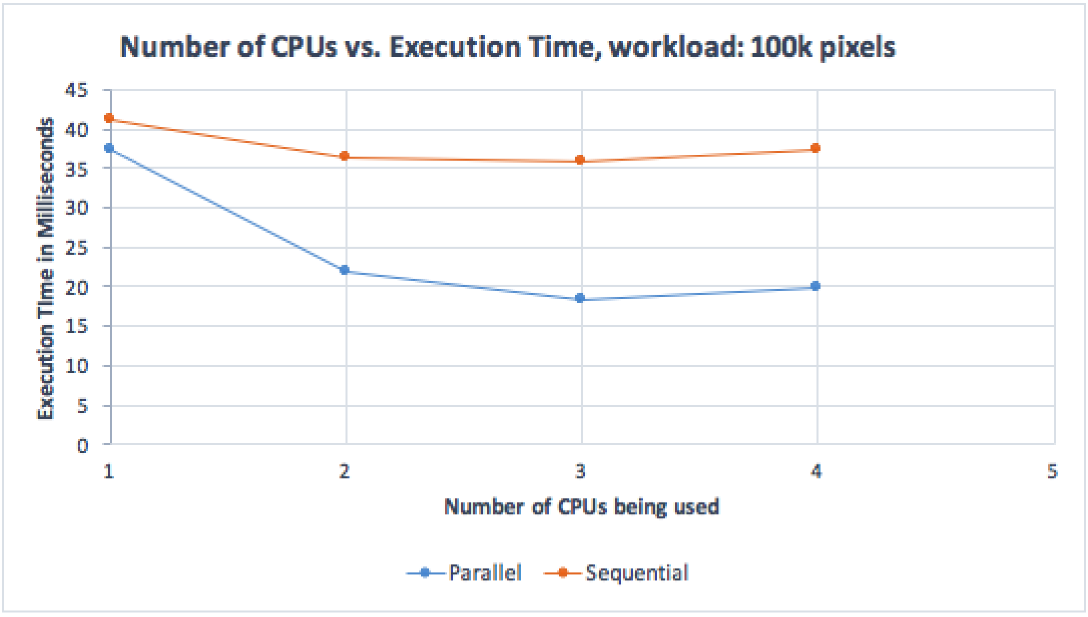
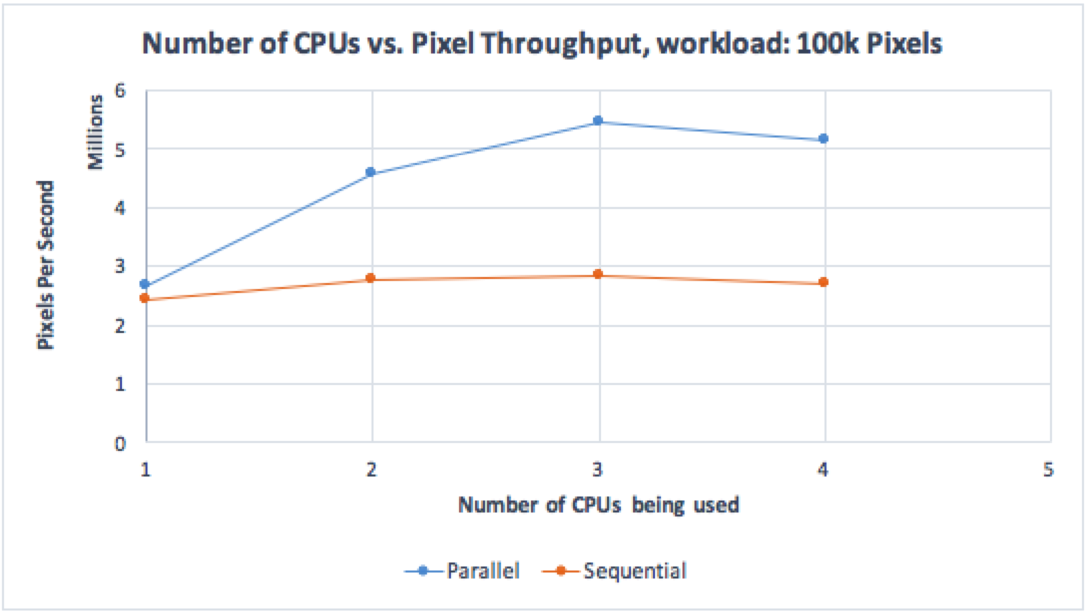


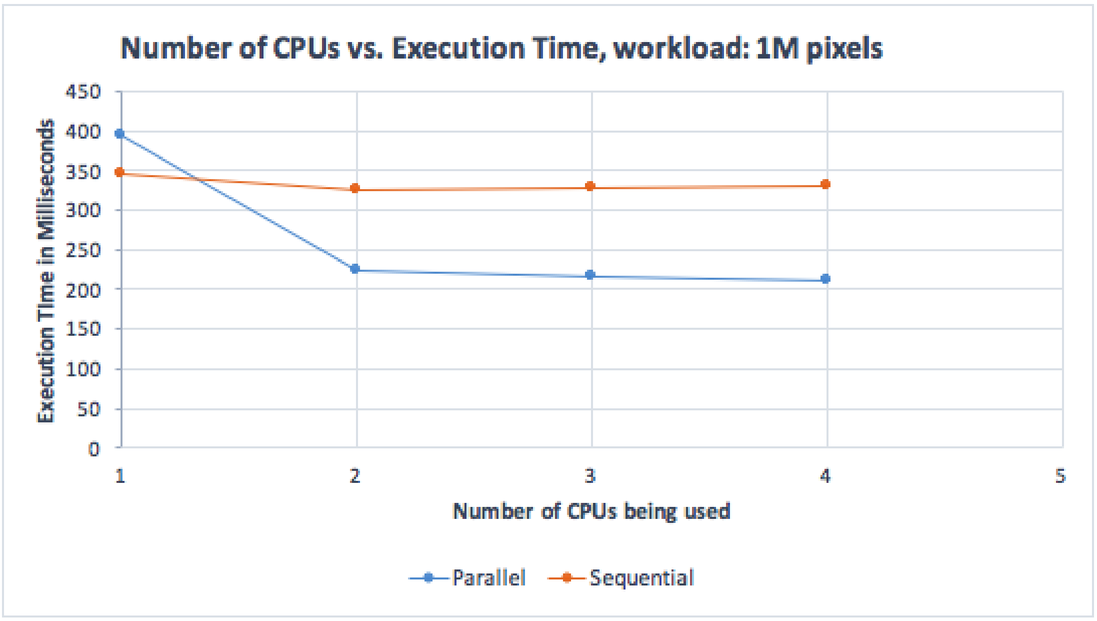
Image Processing Performance Graphs

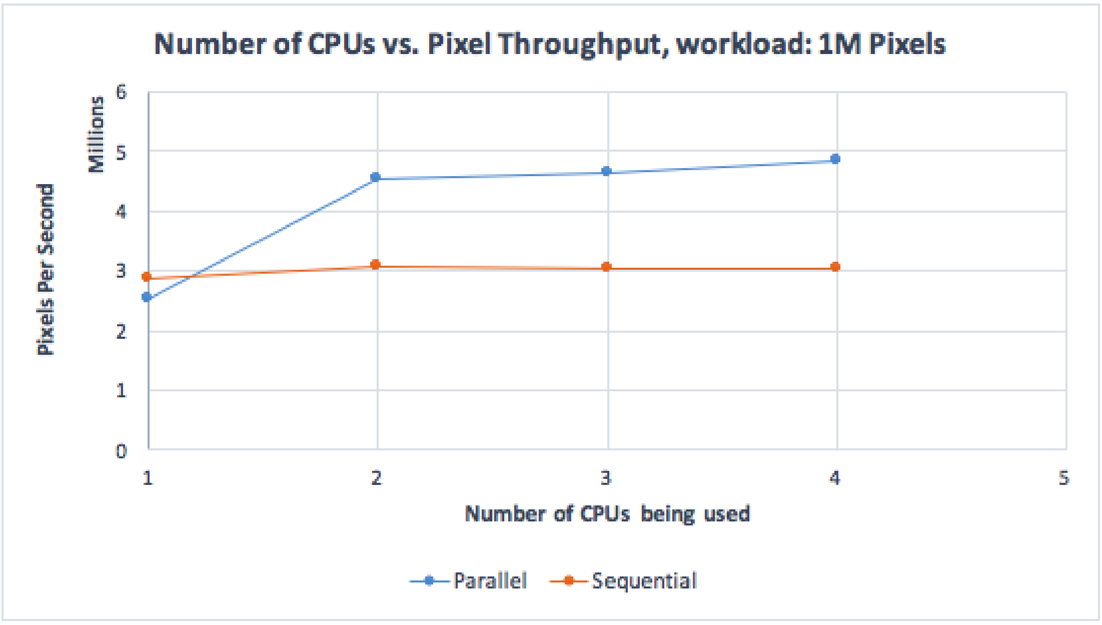
Performance with small data set, image with 100,000 pixels



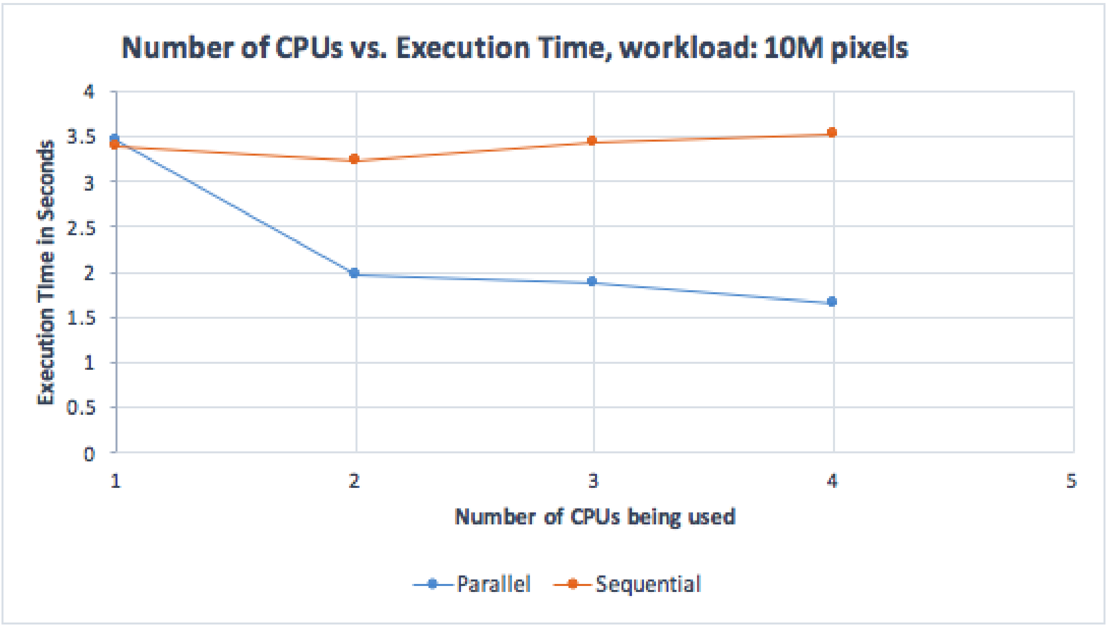


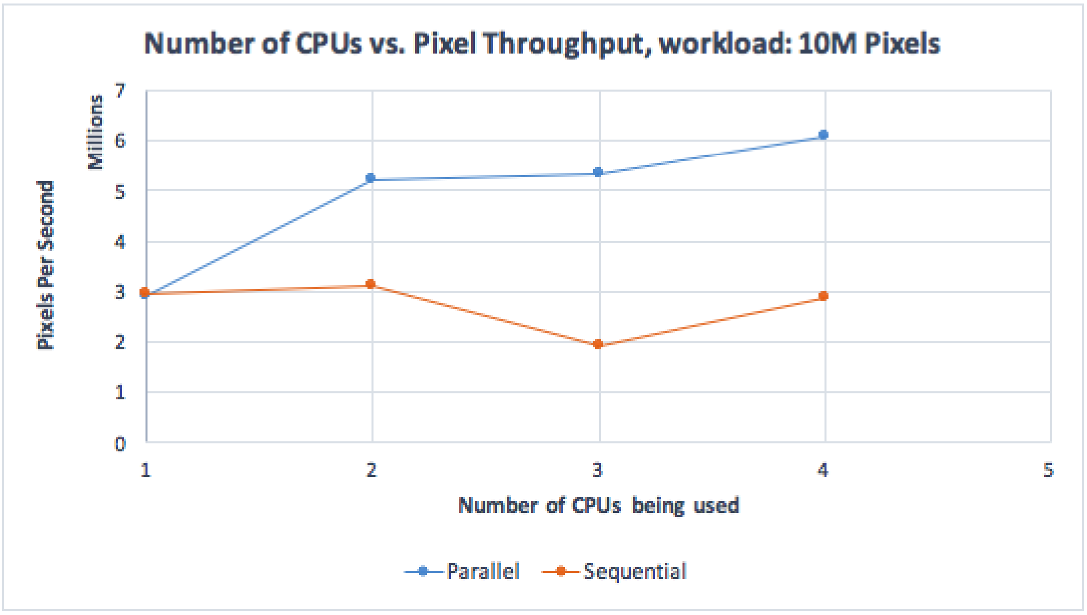
Performance with medium data set, image with 1,000,000 pixels



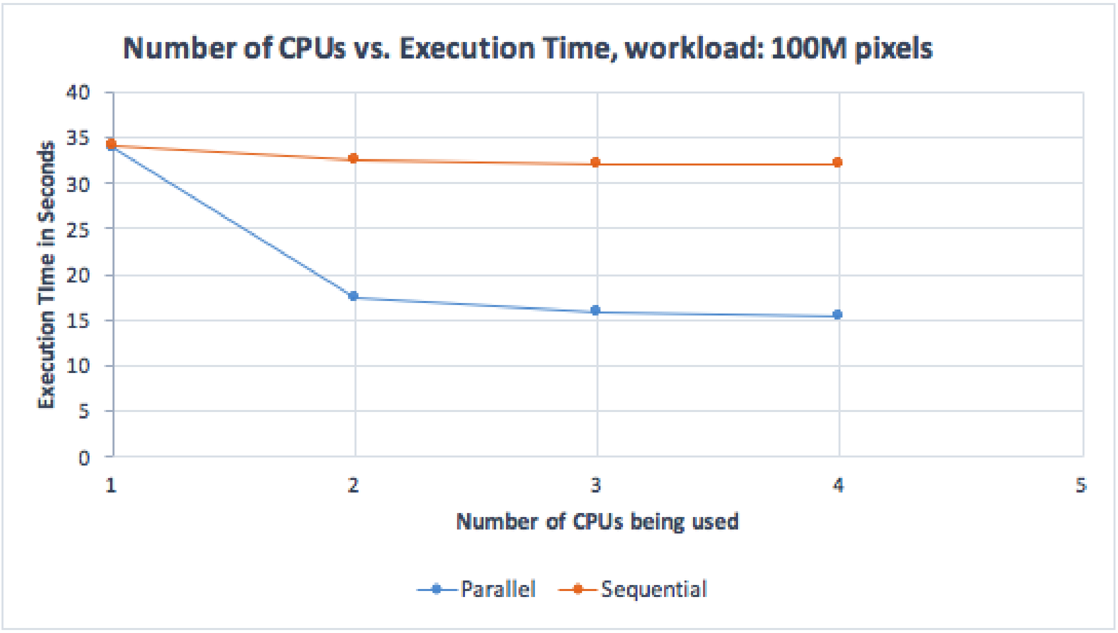


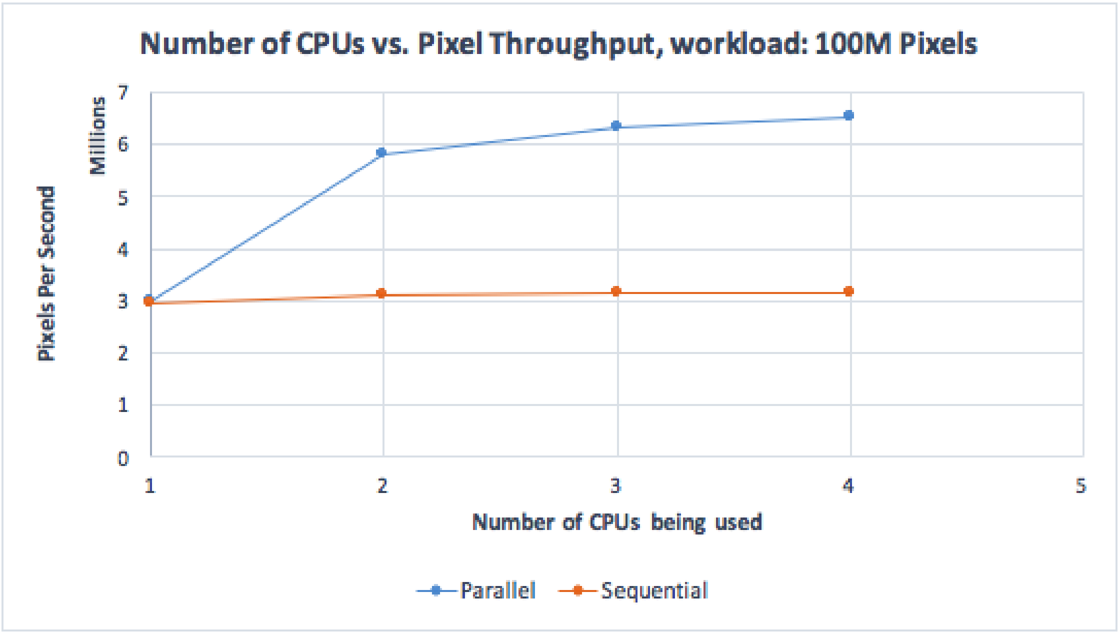
Performance with large data set, image with 10,036,224 pixels





Performance with very large data set, image with 100,800,320 pixels


Number of CPUs vs. Execution Time, workload: 100M pixels
Execution Time in Seconds
Number of CPUs being used
Parallel
Sequential


Number of CPUs vs. Pixel Throughput, workload: 100M Pixels
Pixels Per Second
Millions
Number of CPUs being used
Parallel
Sequential