Project 2 Analysis

Justin Cohler
5/29/2019

Background

Once again in the second project, I found that a context for structuring all necessary parallelization components was hugely helpful in terms of code organization:

```
type Context struct {
    wg             sync.WaitGroup
    scanFilterLock sync.Mutex
    scanCond       *sync.Cond
    filterInCond   *sync.Cond
    qFilter        queue.Stack
    qSave          queue.Stack
    nThreads       int
    readComplete   bool
}
```

This time around, the project requires two queues, a queue to place raw image data after reading the image from file I/O, such that the filterer can work on the image, and a second queue to place finished images once they have been filtered, for storage later on.

On program start, the main thread enters a back-and-forth with a filter controller thread, which signal one another a.) when there is a new message on the queue to filter, and b.) when the filter is finished and the thread is ready for a new image.

Once all images are filtered, the main thread (which is in a Wait() until this point), reads all filtered messages from the *qSave* queue and writes the filtered images to disk sequentially.

Architecture Diagram

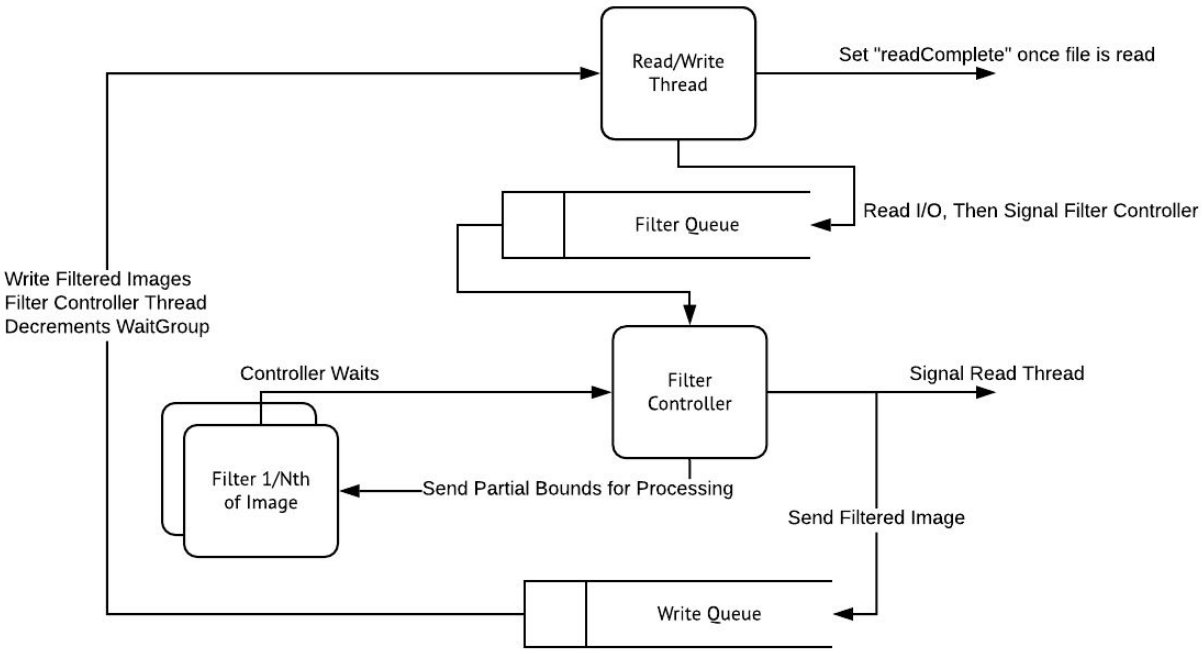


Image Convolution

Each time the filter controller receives an image from the queue, along with the number of threads it can parallelize, the controller splits the bounds of the image into *n* (number of threads) vertical slices, then spawns *n* go routines to process a given set of bounds in parallel. A function called *BlockConvolution* is responsible for this splitting of bounds, and spawns instances of the sub-function *Convolution* with the partial image bounds.

Convolution additionally takes in a "kernel" slice, which is a 3x3 matrix pertaining to the individual convolution required in a given step. This generalized nicely, and allowed various filters to be as simple as a simple slice definition, followed by a call to *Convolution*. The *Blur* example, which follows this convention, is shown below:

```
// Blur applies a blur filtering effect to the image
func (img *PNGImage) Blur() *PNGImage {
    kernel := [][]float64{
        {1. / 9, 1. / 9, 1. / 9},
        {1. / 9, 1. / 9, 1. / 9},
```

```
        {1. / 9, 1. / 9, 1. / 9}}

    return img.BlockConvolution(kernel)
}
```

Functional & Data Decompositions

As described in detail above, the passing of image data from I/O thread to convolution threads, and back to I/O threads for performance sake encapsulated the functional decomposition of this program.

The data decomposition came in the form of splitting each image filter/convolution process into *n* segments, such that a different thread could perform a given filter on the same image in parallel. For the same operation (a given kernel application), multiple processors could parallelize on data, acting as a data decomposition.

Benchmark Specifications

The aforementioned benchmarking was performed on a commodity hardware laptop, with the following specifications:

- **CPU:** Intel Core i7 @ 4GHz
    - 4 cores
    - 8 logical hardware threads (2 per core)
- **Memory:** 16GB DDR3
- **OS:** macOS Mojave

The benchmarking was performed via bash script run several times (to take the average in Excel) running through the cartesian product of all options listed:

- Thread Count: {0 (Sequential), 1, 2, 4, 6, 8}
- CSV Folder {1, 2, 3}

Results

- **What are the hotspots and bottlenecks?**
  The hotspots in this implementation reside in the following spaces:
    - Read/Write I/O - reading and writing from disk is a relatively expensive process.

    - Image Convolution - the image processing that occurs in parallel for each filter is very time consuming compared to, for example, the enqueueing, de-queueing, and csv parsing.

  The bottlenecks are found in two places:
    - Sequential I/O: In both the case of reading files and writing files to/from disk, the program as written performs these actions sequentially. The reason for this is that I/O across threads was warned by the professor to be very expensive. As a result, I dedicated a single thread to do all I/O and suffered the sequential consequences, hoping these would outweigh the slow read/write speeds for relatively few image files.
    - Sequential Filters: In the convolution phase, filters must be performed one after another, as they are not commutative. Moreover, as designed, each image has all filters applied on a given thread. Let's say that many threads had a small number of filters, but one thread had a massive number of filters. The other threads would eventually be halted, waiting for the busy thread to finish. A better design might rebalance to other workers for every filter, to ensure a single heavy image processing does not block other threads.

- **Describe the granularity in your implementation.**
  This was in many ways a coarse grained implementation. The qFilter queue allows for a max queue size of one; each time a message is placed on the queue, the producer sends a signal to the consumer to read, and then waits for a response before continuing to place messages on the queue. Conversely, on the downstream side, all images are placed on the qSave queue before the consumer, which writes filtered images to disk, can begin consuming. This level of synchronization between producers and consumers of each queue is quite coarse.

  While each queue has a lock on the head **and** tail, such that independent reads and writes may occur on a queue, this is not leveraged in practice, as in each case, there is synchronization around when each producer/consumer may act. Because there is no true asynchronous action to the functional side, the implementation is very coarse-grained.

  The only fine grained element to this approach is that the image itself during convolution can be edited in many chunks, asynchronously, and without coordination. This is the sole fine-grained element of this approach.

- **How do increased numbers of threads affect the speedup ratio?**

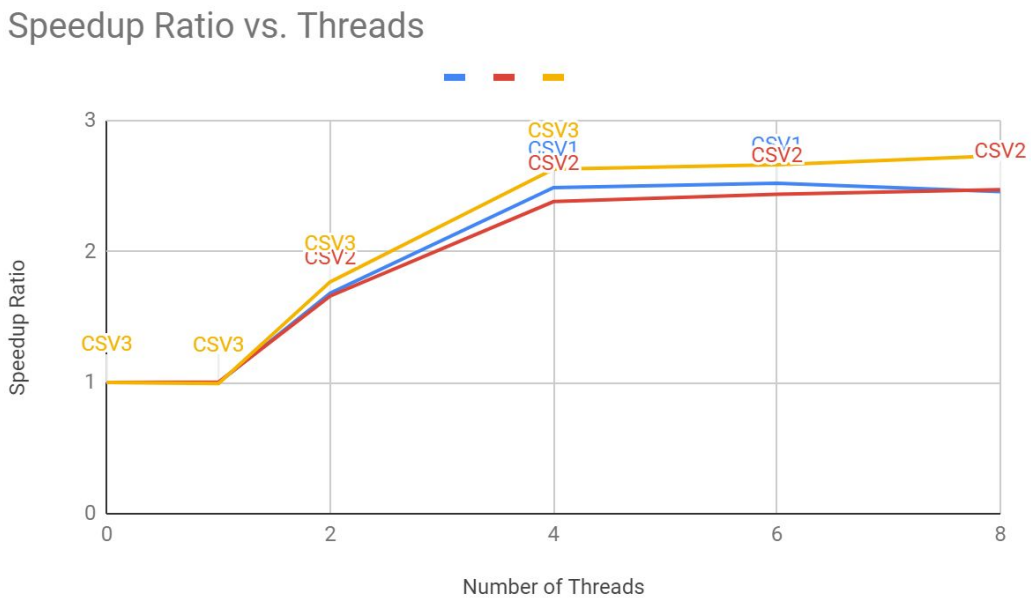
  I ran this program on a four core i7 desktop in the CSIL laboratory in Crerar Library. Plotting the speedup ratio for each of the csv files (containing the same images with

different sets of filter specifications), I found that the speedup ratio hit an asymptote around 2.5-3.0. In general, the filter specifications in CSV #1 were faster than those in CSV #2, which were in turn faster than CSV #3. To give an idea of runtime sequentially, the following metrics highlight base runtimes (in wall-clock time):

- Sequential CSV #1: **`12.156s`**
- Sequential CSV #2: **`78.456s`**
- Sequential CSV #3: **`202.314s`**

What was promising about these results was how close the asymptotic function was for each of these csv files.

The below plot represents these speedup ratios:



- **Does image size have an effect on performance?**

  According to Amdahl's Law, speedup is a function of parallelizability. However, this is not necessarily a linear function, as a given set of images may vary in size here, and their filter sets may vary in complexity. As a result, the proportion of time that encapsulates the parallelizable component of the program may vary for a given image set, and the complexity of filters applied. As shown in the plot above, there are diminishing returns above 4 threads on the current hardware. This may be a result of hardware overloading, but is more likely the result of the images being small enough that the sequential portions of the code overwhelm the amount of speedup we get by parallelizing the images into smaller segments.

In the case where images were very large (running slowly sequentially), the expectation is that this would improve the speedup ratio over smaller images, as this would increase the amount of parallelized work compared to the bottleneck sequential work.