Thread optimization:
Based on the methodology, the number of threads are reduced if does not make reasonable sense. For sharding rows, if there are more threads then rows, the threads is reduced to the number of rows. For sharding Columns, if there are more threads then columns, the threads is reduced to the number of columns. For Work Queue, if there are more threads than the number of chunks needed to be processed(explained further in part3) then the threads are reduced to the total number of chunks. This is to stop threads from being created that would have 0 workload (ex.if they are more threads then chunks the extra threads would have no chunks to process and would instantly finish.)

Zero padding:
Regardless of methodology being used, a zero padded version of the original image is created. This is padded with zeros based on the filter being used (filter 3x3 would be a 1 row and 1 column of 0s before and after the target adding a total of 2 0 rows and 2  0 columns). This is calculated by simply taking the dimension of the filter, and dividing it by 2 and rounding it down(ex  rounded down(3/2)=1, rounded down(5/2)=2). This is done so the filter does not run into any issues when running on the corners and edges of the original image to create the target. The target returned is still the size of the original image.

Apply filter function:
Since the Zero padding was implemented we can now create a new function called applyfilter. This applyfilter does all the calculations for the filter and puts it on the target image , as long as we supply the coordinates(the row and column) that should be calculated for. This made the function much more modular due to the fact we can now call apply filter in all methods to apply the filter. Now  we can simply tell the function which rows and/or columns that need to be calculated (the order of operations such as row first, column first, etc.).We call apply filter on each row*width+col since it is an array and not a matrix where it would be simply [row,col].

Translator function:
The translator function is used to find the index from the zero padded array to the target array since target[row,col] != zero_padded_array[row,col].

# Part 1

Sequentially was implemented with a basic nested for loop that is going one by one through each row and column calling apply filter to calculate the result.  Sequential mode is by far the slowest since it cant execute any threads is often  slower then all methods. There isn't any reason to use this method other then for a quick implementation due to time constraints (needs to be written quickly and simply). Due to no synchronization needed at all, there is no locking overhead and insight necessary to achieve correctness and as such much easier to debug.

# Part 2

First iteration is simply to initialize the local variables min-max in each thread to be the first number processed (calculated from apply filter) since the min-max should not have any defaults(well besides - and + infinity respectively which is more tedious to code in C then this implementation).

**SHARDED_ROWS:**

Each thread is assigned a number of rows based on how many threads and how many rows we have using the formula: height+num_threads-1)/num_threads;.

Each thread then calculates row by row until it runs out of rows to finish. This is done with a for loop to go through each column in a specific row until there's nothing left in the row. Calling apply filter on each row*width+column iteration. Then it goes to the next row assigned until done.

Sharded_row is considerably faster than sequential because it takes advantage of cache line (spatial locality) due to running through continuous blocks of memory(cache lines).

**SHARDED_COLUMNS_COLUMN_MAJOR:**

Each thread is assigned a number of columns based on how many threads and how many columns we have using the formula: (width+num_threads-1)/num_threads; ).

Each thread then calculates column by column until it runs out of columns to finish. This is done with a for loop to go through each row in a specific column until there is nothing left in that column. Calling apply filter on each row*width+column iteration. Then it goes to the next column assigned until done.

Sharded_column_column_major is considerably faster than sequential because it takes advantage of temporal locality. (The column values are being called multiple times in a short time frame).

**SHARDED_COLUMNS_ROW_MAJOR:**

This was implemented very similarly to sharded_rows except now the for loop for height was the other loop instead of the inner loop , such that the threads prioritized doing each col in a row before going to the next row.

This is similar to sharded_rows where it takes advantage of local spatiality through cache-lines to improve performance. This is also similar to sharded_columns_column_major in that also takes advantage of temporal spatiality through the use of calling column values frequently in a short time frame.)

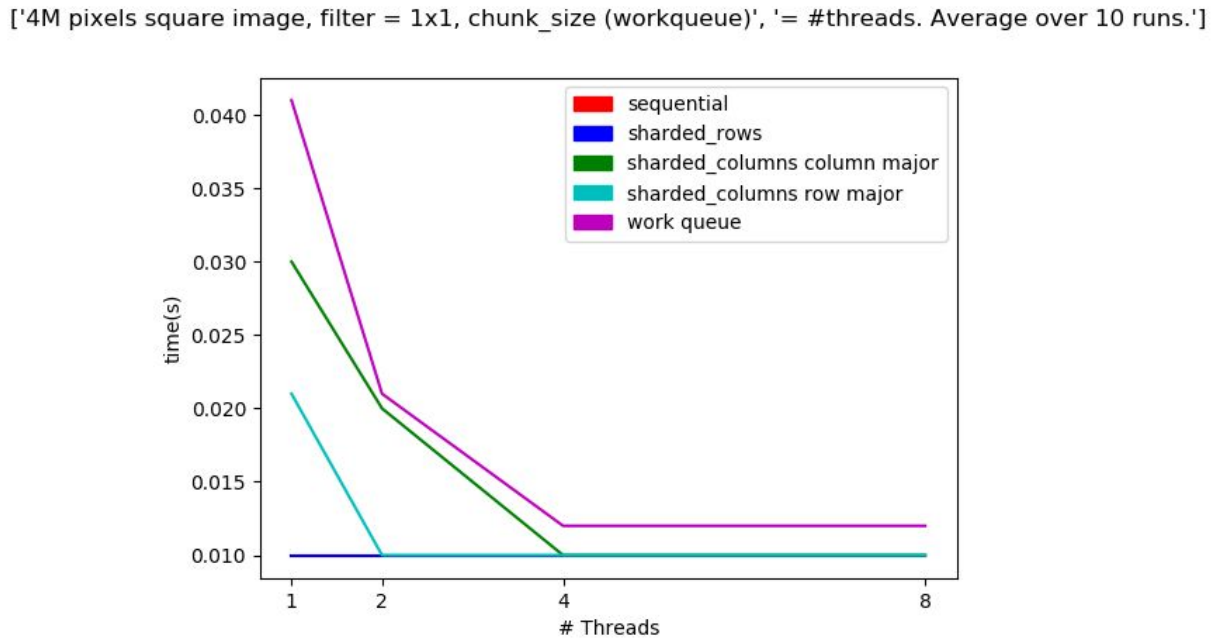
# Part 3

Work_queue was implemented using chunks(n x n). First, the number of total chunks was calculated using the number of times a chunk would fit in the height (number of rows) and the number of times a chunk would fit in the width(number of columns). This would give us the total number of chunks need to be processed as total chunks=(chunks per height)*(chunks per width).Next the chunks were assigned to originally take the original thread number. The threads automatically updated the shared variable with locks(last chunk) when a chunk is assigned to a thread. This last chunk variable indicates the last chunk that was assigned. Once a thread has finished it will call on the last chunk variable with locks to get the next chunk(last chunk+1) that needs to be processed. This is done in a while loop so once the last chunk = total chunks, means all the chunks have been processed and to exit the loop. In the edge case that the chunk size is larger than the original image(very small), there is an if statement to make sure the row and column does not call apply filter for rows and columns greater than height and width respectively.

Work_queue is the most difficult to implement. Work_queue minimizes this idle time heavily by breaking the tasks up in chunks assuming the chunks are sized proportionally to the workload. In such a case, the threads will keep continuously picking up new chunks from the work_queue until all chunks have been sent to be processed. This can heavily reduce idle time since the most amount of work left to be done would be the same size as max_threads-1*chunk size(n by n) opposed to thread 1 finishing in the first case and the other 31 threads still need to finish (which would most likely be

much larger up to (max_threads-1/max_threads)*(original image size)). This reduction in idle time should increase the performance of work_queue. However, as stated previously the work chunks must be appropriate for the size of the image, otherwise the locking overhead would be massive for the speed incase(1 by 1 chunk size for a 100 million integers is quite substantial). Additionally if the chunks are too big, then you are lowering the effectiveness given by the Work_queue implementation as both idle time of the threads should increase and each thread would be doing more work possibly lowering the parallelism ( 10 threads processing 500 by 500 blocks will likely finish faster than 5 threads processing 700 by 700 despite the second case having fewer calculations 2.5M vs 2.45M). In addition, work queue cannot use local and temporal spatiality to the same degree as the previous methods have done (not including sequential). This will hurt significantly performance as the images get larger requiring more locks if the chunk size is small having minimal spatiality.

# Part 4 Analysis:

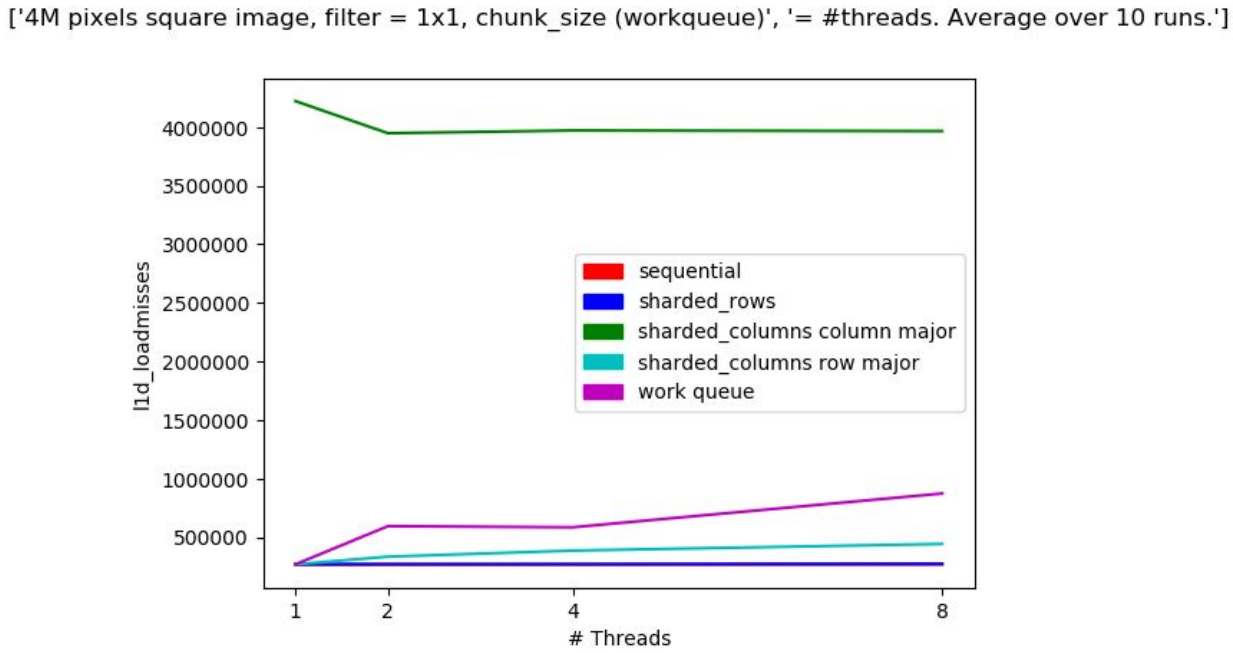



**Changing thread count:**

This first image is running the identity filter and seeing how each method compares. The work queue due to the significant locking overhead is causing terrible performance in comparison to other methods. However as we increase the number of threads , the speed of work queue improves tremendously seeing the most performance gain yet is it

still significantly slower than the sharded methods. This is mainly due to the spatiality that the methods taken advantage off in contrast to work queue. Surprisingly, sequential is one of the fastest when using 1 by 1 filters.

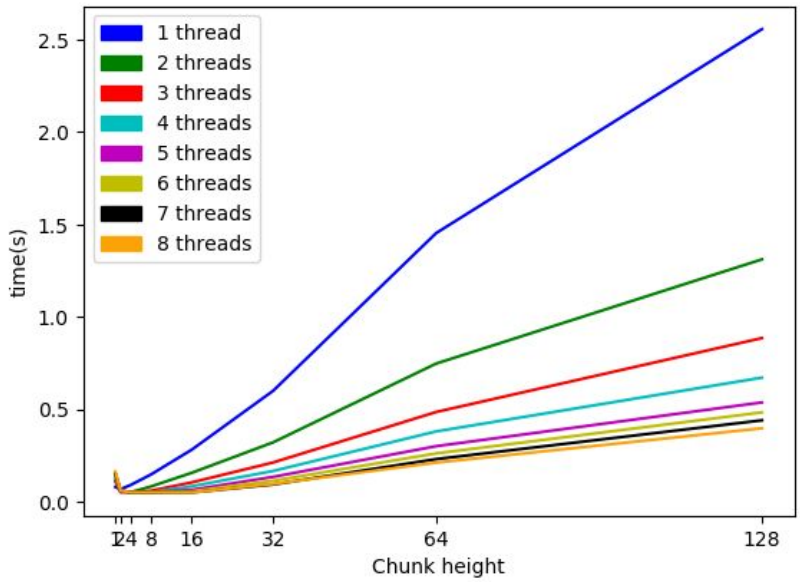
**L1-Cache-misses:**




Here as expected the thread count isn't significantly improving the L1 due to being unable to read from the same cache line. This is where the local spatiality of sharded_column_row_major is vastly most effective then sharded_column_column_major.  This lack of local spatiality found in other methods is what is causing sharded_column_column_major to have significantly more misses since it is unable to take values from the same cache-lines causing more  L!-cache-line misses. Work_queue suffers slightly due to the fact it is not using local spatiality as effectively as sharded_rows and sharded_colum_row_major but it is still using some local spatiality which is why it still has significantly less L!-cache-misses than sharded_column_column_major.

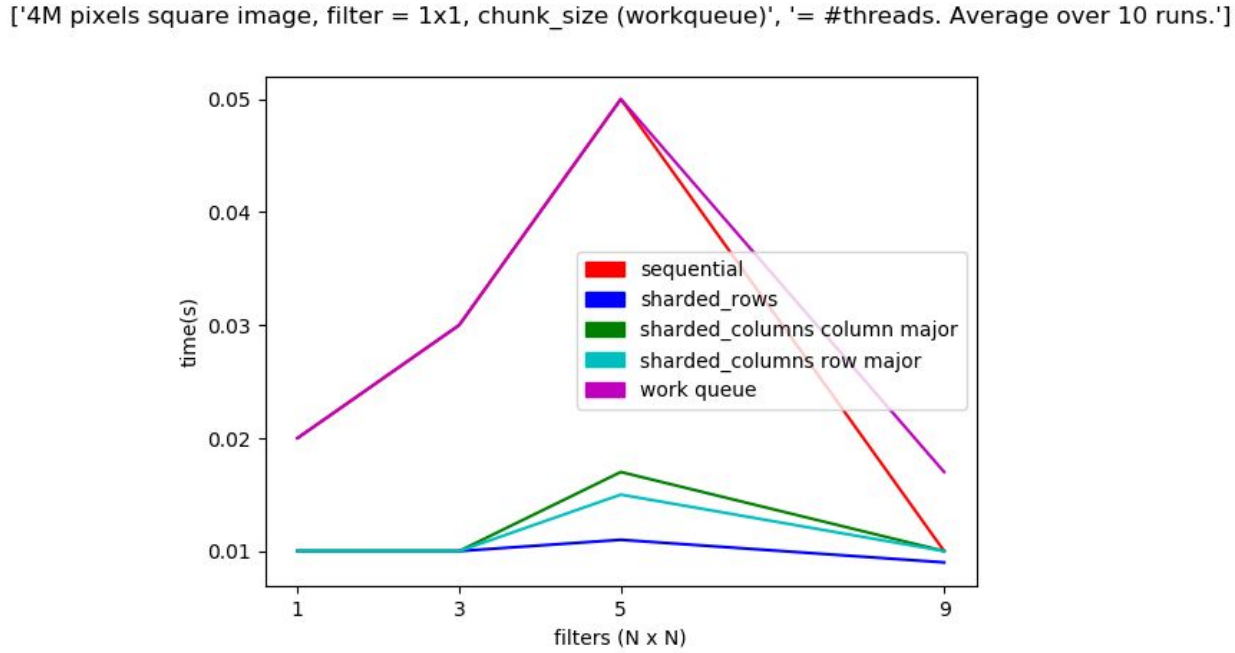
**Work Queue speed affected by threads:**

['4M pixels square image, filter = 9x9, chunk_size (workqueue)', '= #threads. Average over 10 runs.']




Here with all other factors being constant, increasing the number of threads improved the speed linearly where the number of threads is directly impacting the speed of the work queue. Work_queue is almost 8 times faster when running with 8 threads in contrast to 1 thread and is 4 times faster than when running with 2 threads. This makes sense as Work_queue works to actively minimize idling time of all the threads available thus by using this queue we can find speed improves when we are able to use a very large number of threads. In addition, we can see how impactful increasing the chunk height decreases the performance of the Work_queue. This is due to the Work_queue working more sequentially the larger the chunk size is especially with 1 thread it is unable to minimize idle time since their is only one thread that is always being used during the execution of the program.

**Different Filters on method's performance:**

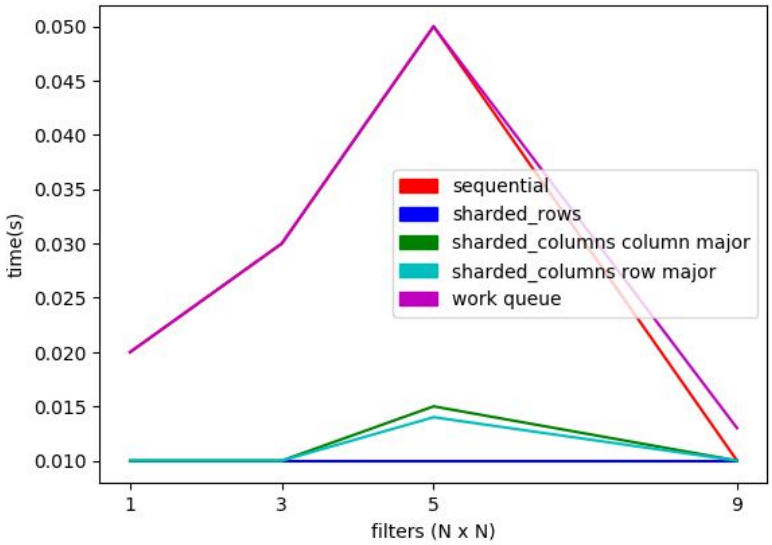



Using 8 threads(# of cpus)

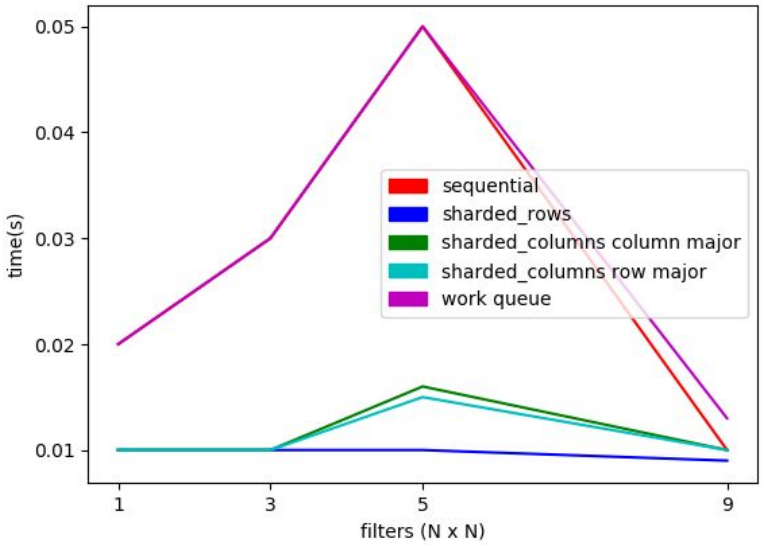
This image shows how the methods interact as we change the filter size. All the filters seem to slow down tremendously as we use the filter 5x5. In addition, the work_queue filter seems to be slower then even sequential as we use filters 5x5. This could be from additional overhead needed as seen in figure 1. This could mean that with these bigger filters the work_queue is doing more work with each thread without taking advantage of spatiality of having bigger and bigger filters(since the numbers would be used continuously in memory and the bigger filters would have larger column usage meaning more frequent use of the same variables). Once we hit the filter 9x9 the effects of having these spatiality is working to improve performance as we are taking larger and larger chunks. This Work_queue is significantly slower than the sharded_methods due to the locking overhead and being unable to use the spatiality as effectively despite being able to minimize idle time with 8 threads (which as seen before significantly improves Work_queue's performance linearly to the number of threads.)

## Unique case:

['100M pixels 1column.pgm image, chunk_size (workqueue) =', '#threads. Average over 10 runs.']




['100M pixels 1row.pgm image, chunk_size (workqueue) =', '#threads. Average over 10 runs.']



This was a unique case of 2 images, both with the size of 100M pixels with 1 column and 1 row respectively. Interestingly, this did not affect the performance as one would expect for sharded_rows and sharded_columns_column_major. It was originally thought to have significantly worse performance as only 1 thread could be run in both those scenarios as sharded rows running 1 row and sharded columns running 1 column could only use 1 thread. However, it seems that this does not impact performance as one would expect with both graphs being nearly identical in nature. This is possibly due to the fact that it is a 1D array that creates the images but also in consideration that the calculation at each iteration would be at most one. Hence it is essentially running in sequential mode but taking advantages from the spatiality nature of each method(ignoring work_queue). Once again work_queue is the worst method as it is either the same speed or slower than sequential despite having 8 threads.

**Conclusion:**

Sequential is the easiest to implement and will typically be the slowest method. It is unable to effectively use threads and as such will take the longest amount of time if a program can be executed in parallel with threads typically.

Sharded rows is able to take effective advantage of local spatiality through cachelines. This will give significantly increased performance if used in correct to get chunks of memory continuously.

Sharded columns column major takes advantage of temporal spatiality through the fact the same values in memory is being used repeatedly in short succession. This will also allow for significantly increased performance if used correctly where it saves a chunk of memory that needs to be used again. This is typically not as fast as Sharded_rows due to cache-line's performance.

Sharded_columns_row_major tries to take advantage of both temporal and local spatiality to maximize performance. This is can reach very close to sharded_rows performance while still calculating column by column. This can be useful in certain applications that need to calculate per column.

Work_queue is a very interesting case. The difficulty to implement especially including having to use locks(locking overhead) and the terrible performance(even worse than sequential) in situations where it is used non-ideal situations can be heavily discouraging. That said the scalability with Work_queue and its ability to minimize idle time on the threads it is being run on can be extremely effective in ideal situations. The

work_queue will ensure very low idle times, and is capable of using much more threads than all other methods in ideal situations. Sharded_rows limited by rows, sharded_columns limited by columns , while work_queue is limited by the number of chunks which is often times significantly larger than either of those options.