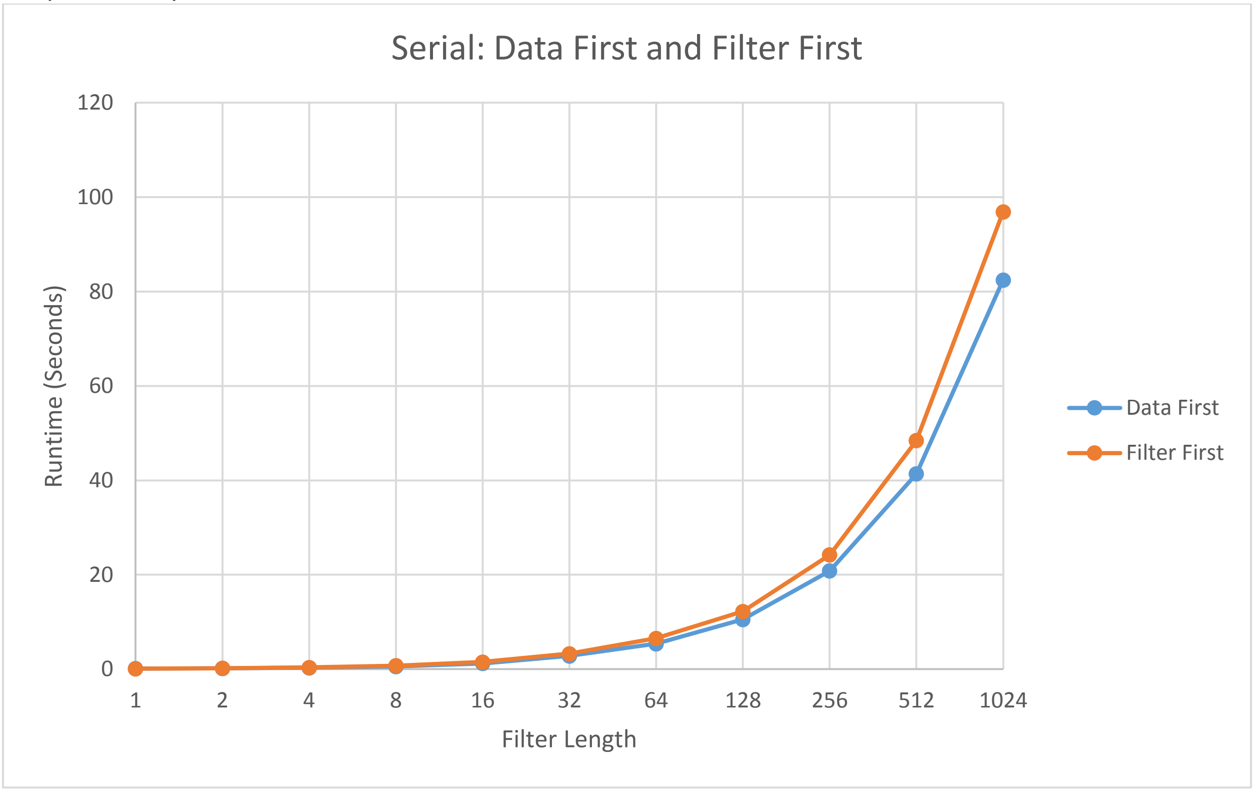
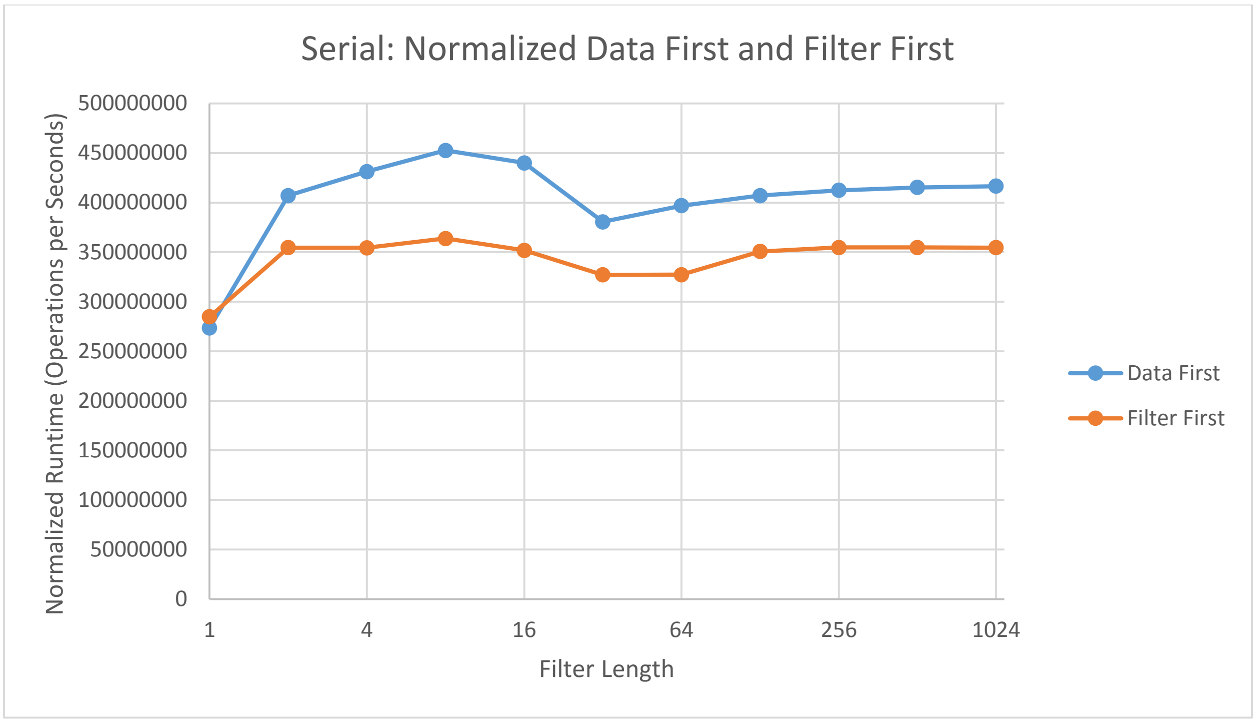
Stephanie Chew
schew4
schew@jhu.edu
Parallel Programming - Spring 2015

Loop Efficiency





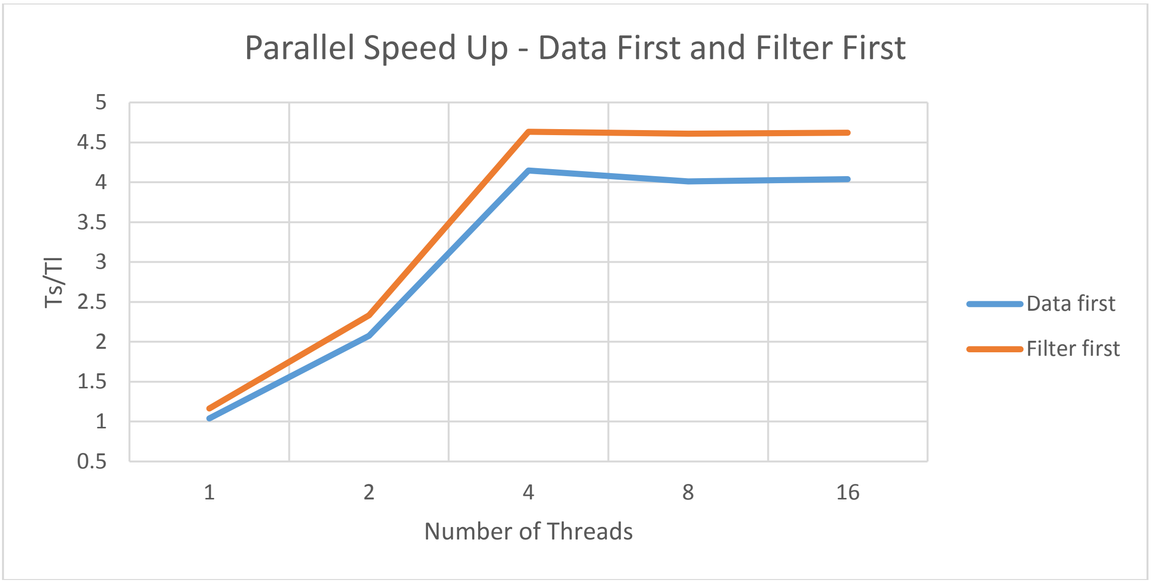
1c. It's more efficient to have data first because there are fewer memory accesses. The cache accesses rows in memory at a time, so having a smaller row to fill is more efficient, i.e. since the filter array is smaller, you would have to fill the cache fewer times. Whereas, having data in the inner loop would mean that there would be a significantly lot more accesses to memory to fill the cache. As a result, having more memory accesses would take up more time and hinder performance, so it's better to have data in the outer loop.

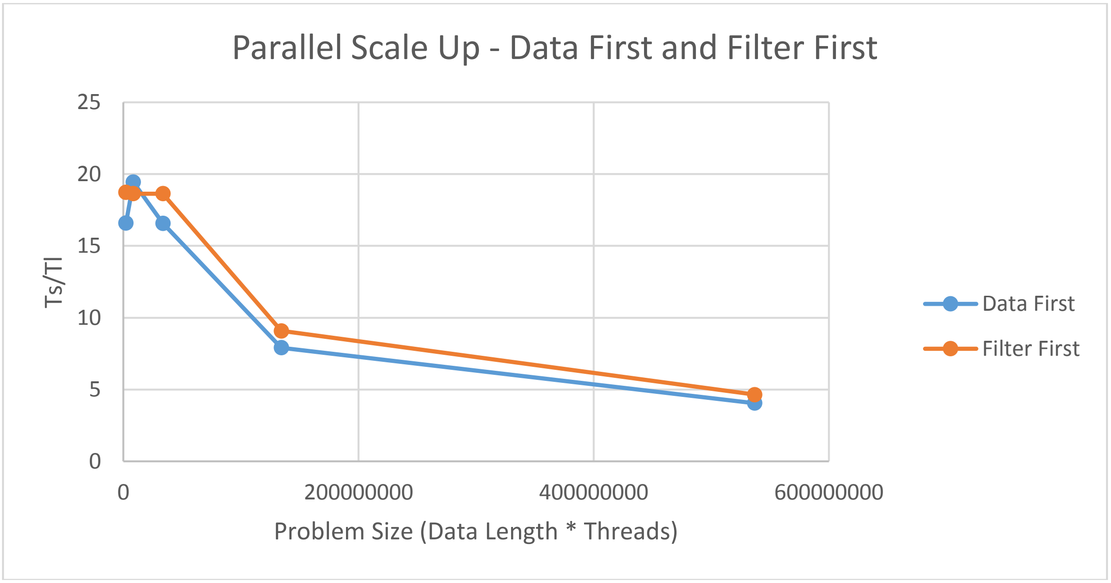
1d. There is a trend that the performance is relatively poor for smaller filter lengths and improves for larger filter lengths, but the performances for the larger filter sizes are about constant. We see this because there is a constant cost that has a big impact in the time it takes for the smaller filter sizes to run, so the overhead is large when the filter length is small. On the other hand, the cost is still there for the larger filter sizes but it has less of an impact since the majority of the time is devoted to the operations so the overhead has a smaller impact.

Loop Parallelism

1b. Here we have a sublinear speedup because as the resources, or number of threads, increases, the speed plateaus out so adding more resources won't make the performance any better. It initially behaved linearly, which is ideal, but began to tail off after 4 threads as it reached the maximum number of threads where any additional ones will no longer improve performance and other factors begin to hinder performance. This program was run on a c3.2xlarge instance which has 8 cores. However, since we tried to make it run on 16 threads, there are more threads than cores and so there is overhead context switching.

1d. We saw sublinear scaleup because as the problem size (or data length) increased with the number of threads, the performance dropped as ideally you would want it to be constant. It initially was about constant, which is ideal, but began to tail off after 4 threads, and ran on a c3.2xlarge instance with 8 cores. In reality, adding more resources, or threads, will only impede performance. However, one anomaly we saw is that even though the instance has 8 cores, in theory, there should be a tail off around 8 threads.





2a. Comparing the performance of data-first and filter-first, we see that they are both evenly matched, but with data-first slightly faster as you still have more memory accesses with filter-first. However, running the program in parallel only has a difference that's slightly smaller than running the program serially.

2b. Based on the calculations using Amdahl's Law, when running on one thread, 1.0379 = 1/(1-P+P/1), so P is approximately 0.

On two threads, 2.077 = 1/(1-P+P/2), so P = 1.004

On four threads, 4.146 = 1/(1-P+P/4), so P = 1.0117

On eight threads, 4.008 = 1/(1-P+P/8), so P = .8627

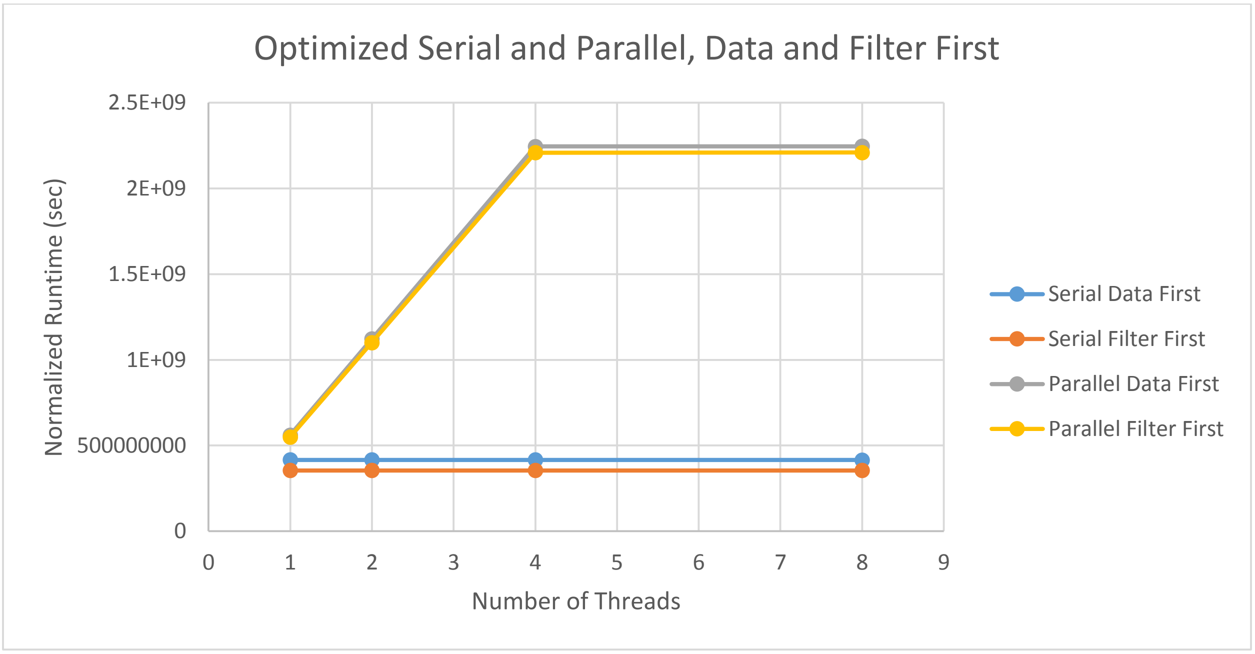
However, we see that running the program on eight threads does not give a consistent P as expected due because of the anomaly in performance noted above.

2c. We see nonideal speedup because of interference. As we increase the number of threads, there are more accesses to the memory and so we see this is one of the limiting factors in performance time. Also startup costs factor into the performance as there is always a cost to initiate processes and creating and destroying threads. We saw that as there are more threads, the startup cost is greater since there are more to create and terminate.

An Optimized Version

1a. For this Section, I applied loop unrolling, which significantly decreased the time. Unlike in the pdf, the inner loops were advanced by 4 instead of 2. It improved performance because it reduces the number of loop iterations by 1/4th and allows the compiler to pre-calculate offsets, so this cost is taken from runtime and moved over to compile-time. In addition, it was run on 4 threads because having 8 threads hinders performance. We can see the significant improvement that loop unrolling has in the graph below. collapse(2) improved the performance by about .3 sec, which is not a significant amount, but was included anyways. It is supposed to help collapse multidimensional loops down to one.

1b. Loop fusion is not applicable, because the bounds of the loops are not the same, loop fission is not useful. Loop tiling hindered the performance when implemented with loop unrolling, but it reduced runtime by itself, but the impact was not as significant as loop unrolling, so only loop unrolling was included and loop tiling was removed.


Optimized Serial and Parallel, Data and Filter First
Normalized Runtime (sec)
2.5E+09
2E+09
1.5E+09
1E+09
Number of Threads
Serial Data First
Serial Filter First
Parallel Data First
Parallel Filter First