Parallel Programming HW 1

# Loop Efficiency

* 1. Plot the runtime as a function of filter size (don’t forget to label axes and units).

|  |  |  |
| --- | --- | --- |
| FilterLength | SerialFirst | DataFirst |
| 1 | 0.197995 | 0.234187 |
| 2 | 0.310235 | 0.288046 |
| 4 | 0.620462 | 0.522065 |
| 8 | 1.241307 | 1.275145 |
| 16 | 2.481903 | 2.791405 |
| 32 | 4.963594 | 4.233707 |
| 64 | 9.923281 | 9.126069 |
| 128 | 19.85476 | 17.82051 |
| 256 | 39.771 | 35.20575 |
| 512 | 79.51974 | 69.90421 |
| 1024 | 158.9741 | 139.3948 |

* 1. Normalize the runtime to the filter length and plot the normalized runtime, i.e. number of operations per second.

|  |  |  |
| --- | --- | --- |
| Normalized Runtimes | SerialFirst | DataFirst |
| 1 | 338943072 | 2.87E+08 |
| 2 | 432632450 | 4.66E+08 |
| 4 | 432638376 | 5.14E+08 |
| 8 | 432504539 | 4.21E+08 |
| 16 | 432628528 | 3.85E+08 |
| 32 | 432646962 | 5.07E+08 |
| 64 | 432817260 | 4.71E+08 |
| 128 | 432638627 | 4.82E+08 |
| 256 | 431969740 | 4.88E+08 |
| 512 | 432090683 | 4.92E+08 |
| 1024 | 432268515 | 4.93E+08 |

* 1. Is it more efficient to have the data or the filter in the outer loop? Why?

The performance of the DataFirst method is better, this is essentially due to how caching works. When the dataFirst loop runs first, for every iteration of that loop the for loop for filter runs. On the first iteration of the filter loop, the filter list at 1 is accessed, and when that happens, the entire filter list (or at least the first 128 blocks) is put from memory into cache. This means that when the second iteration of the filter for loop runs, the value of filter list at 2 will be taken from cache instead of RAM, making the process much quicker. This process if very similar to accessing a two dimensional array in a nested for loop.

In essence although this caching process happens for data first, there are more cache misses as the total filter can be put into cache whereas the total data cannot hence there will be more cache misses for filterfirst and it will be slower.

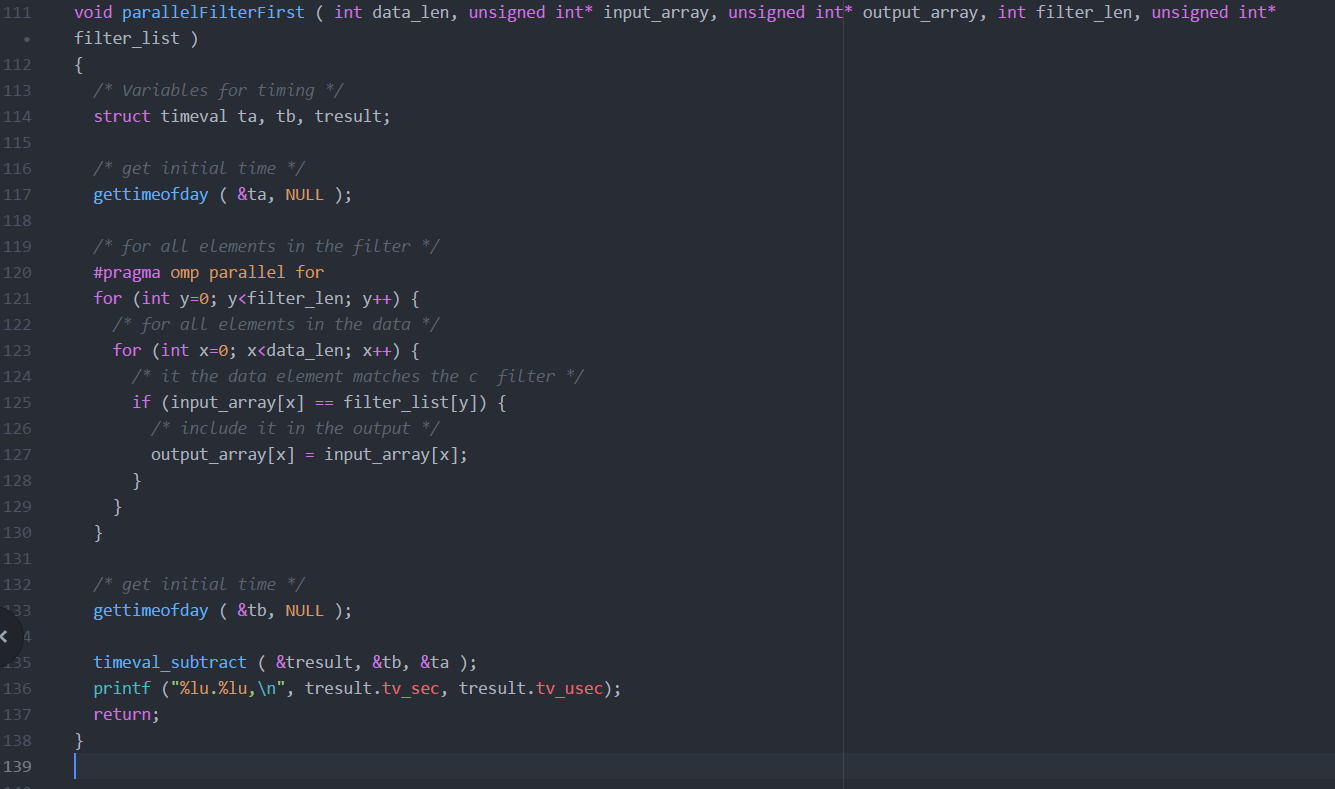
* 1. How does the relative performance scale with the filter size? Explain the trend.

As the filter size increases, we will see an increase in performance as more and more data will be able to fit in cache, leading to less cache misses. However as we reach about higher filter sizes, this performance gain will tend to decrease and level out as the filter can no longer fit into cache completely and there will have to be at least one cache miss in order to read the whole filter length.

# Loop Parallelism

* 1. A. Create a function parallelFilterFirst that is the same as b. serialFilterFirst except that it has the #pragma omp parallel for around the outer loop.

See line 111 from file filter.c.



* 1. B. Create a function parallelDataFirst that is the same as serialDataFirst except that it has the #pragma omp parallel for around the outer loop.

See line 142 from file filter.c.



* 1. A. Generate speedup plots for 1,2,4,8, and 16 threads.

B. Describe the results.

The speedup for both datafirst and filterfirst were very similar, so we will examine them as one. The speedup was linear when the number of threads were between 1 and 4, however, after that the speed up slowed to being almost null and very slow. This tapering that took place past 4 threads, is due to the hardware of our EC2 Instance. Our machine is a 4 core machine, therefore any thread that we try to use over 4 will not be efficient. Indeed although out machine does have 2 threads per core, scheduling multiple threads on a core does not yield drastic improvement. Moreover any thread we try to use over 8 threads will just be scheduled serially and rendered pseudo useless.

* 1. Parallel performance.

1. Which version of the parallel code was faster? Why?

The DataFirst code was very marginally faster than the filterfirst one. This is due to same reason as for serial functions. Cache Memory access is more optimal for dataFirst than for filterfirst.

1. For the faster, estimate the value of p in Amdahl’s law based on your results for 1 to 8 threads.

Let n be the number of threads, S be the speedup for that thread and *p* be Amdahl’s number.

We know the formula for Amdahl’s number is :

So we have

* For 1 threads:
* For 2 threads:
* For 4 threads:
* For 8 threads:

1. To what do you attribute any non-ideal speedup? To startup costs, interference, skew, etc? Explain your answer.

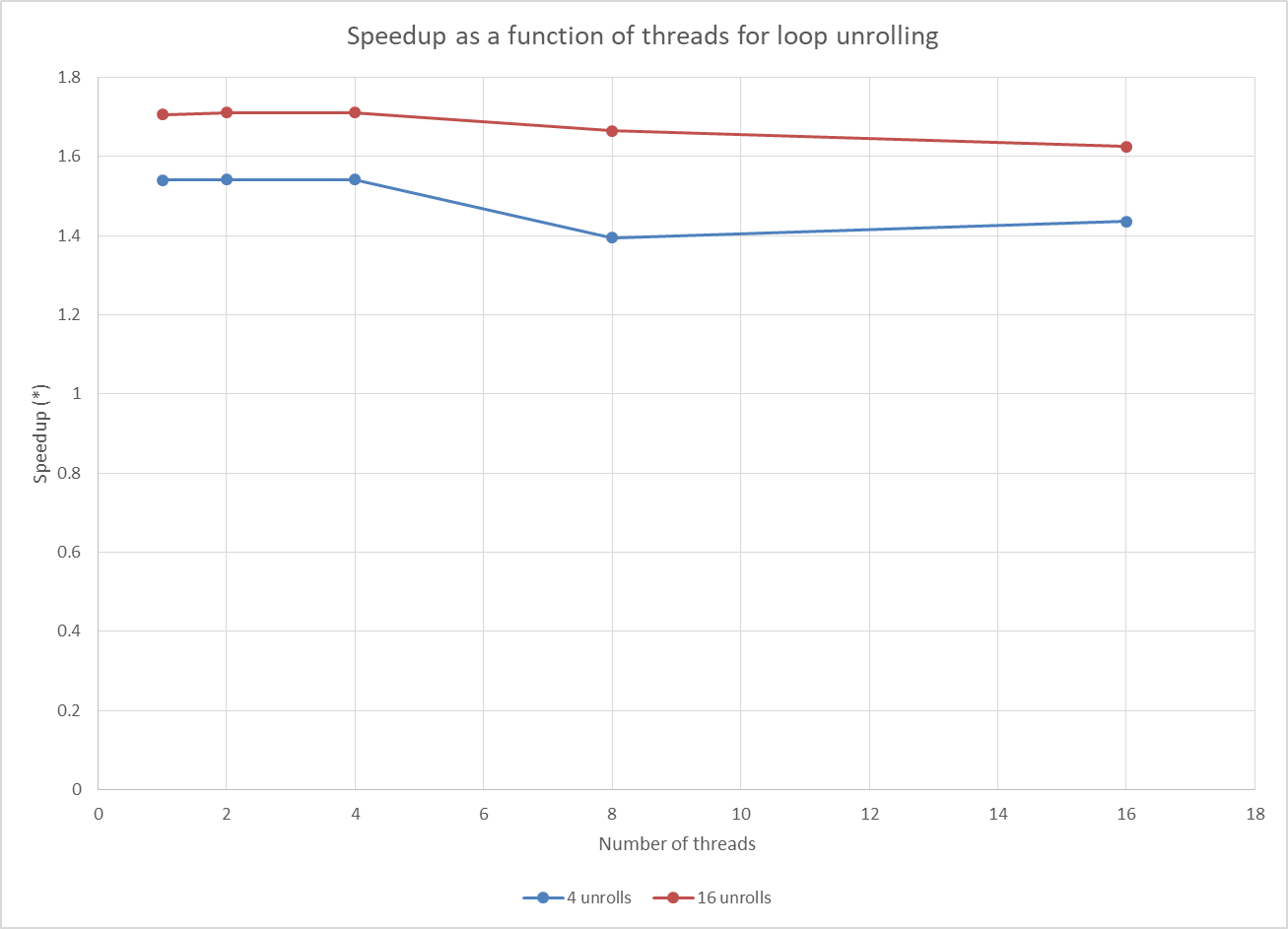
We can attribute the non-ideal speedup to startup in part as OpenMP has to unroll the loops and then schedule which threads to run which tasks. This is definitely a part of why we have non-ideal startup however the biggest problem is probably interference as the threads will interfere with each other when accessing memory which will cause a drop in performance and speedup.

# An Optimized Version

1. Loop Unrolling

See line 174 in filter.c

For loop unrolling, we unrolled the loop 4 times and 16 times. We then ran these two functions at 1, 2, 4, 8 and 16 threads and compared the normalized operations per second to the non-optimized loop as well as the speedup of the optimized loop:



*Here we can see the speedup factor that loop unrolling gives us.*



We can obviously see that unrolling the loop optimizes the parallel datafirst function. This is due to the fact that if we unroll a for loop x amount of times, the new loop will have x times less iterations to go through. Since the small loop gets unrolled, there is less startup cost for one, and therefore less overhead when OpenMP has to schedule the iterations.

Moreover there will be more sequential checks and less jumps if the loop is unrolled which will reduce the operational runtime.

1. Custom Scheduling

See lines 278 and 310 in filter.c

For Custom Scheduling, we chose to use a chunk size of 4 as we tested different chunk sizes and could see that the chunk size did not affect the runtime of dynamic or static scheduling.

Here we have graphs for speedup and runtime of our dynamic and static scheduling compared to non-custom scheduling:

We can observe that there is barely any speedup for dynamic and static custom scheduling.

Why is that? Dynamic Scheduling is used when we want openMP to dynamically distribute the amount of work among the cores, whoever when the operations themselves all take the same time, dynamic scheduling does not really hasten the process in anyway. Furthermore, for every iteration of the loop there is more overhead with dynamic scheduling as OpenMP has to resend new values of the loop variables to each thread and this can slow down the entire process.

As for Static scheduling, we understood that the default openMP scheduling method is in fact static. Therefore it is normal that we did not observe any speedup.