Parallel Programming OpenMP

Detian

February 22, 2015

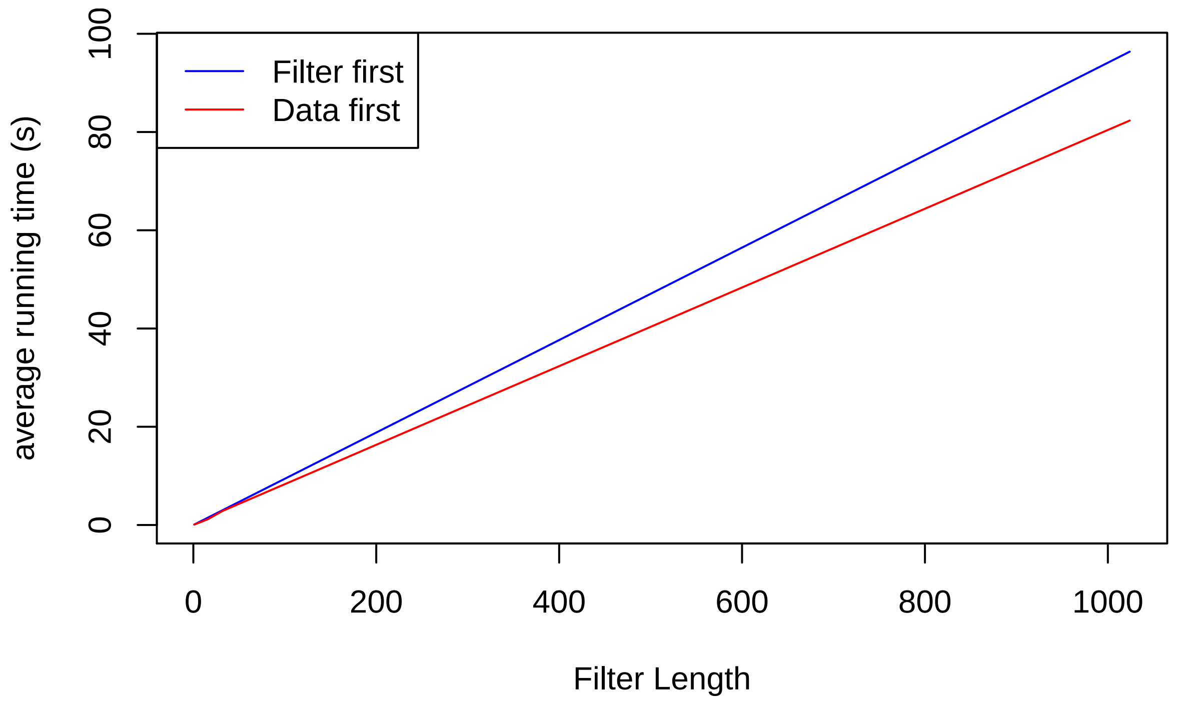
Hardware Information

AWS c3.2xlarge instance with 8 virtual CPUs (4 physical cores) of 2800.080 MHz and 14 GB memory.

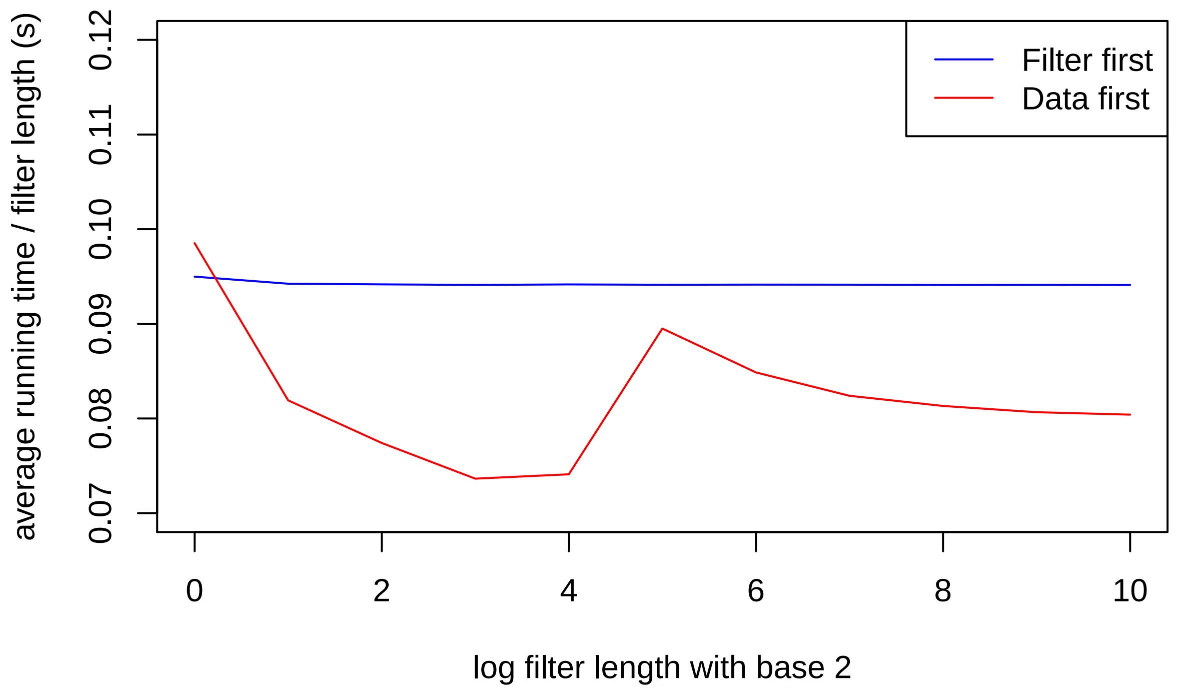
Loop efficiency

1. (20 pts) Evaluate the performance of the functions serialFilterFirst and serialDataFirst that loop over the arrays in different orders as a function of the filter length. You should select a constant value for the data length that allows each experiment to complete in a reasonable, but non-trivial amount of time. You should vary the filter length as 1,2,4,8,16,32,64,... (at least to 1024).

- Plot the runtime as a function of filter size.



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

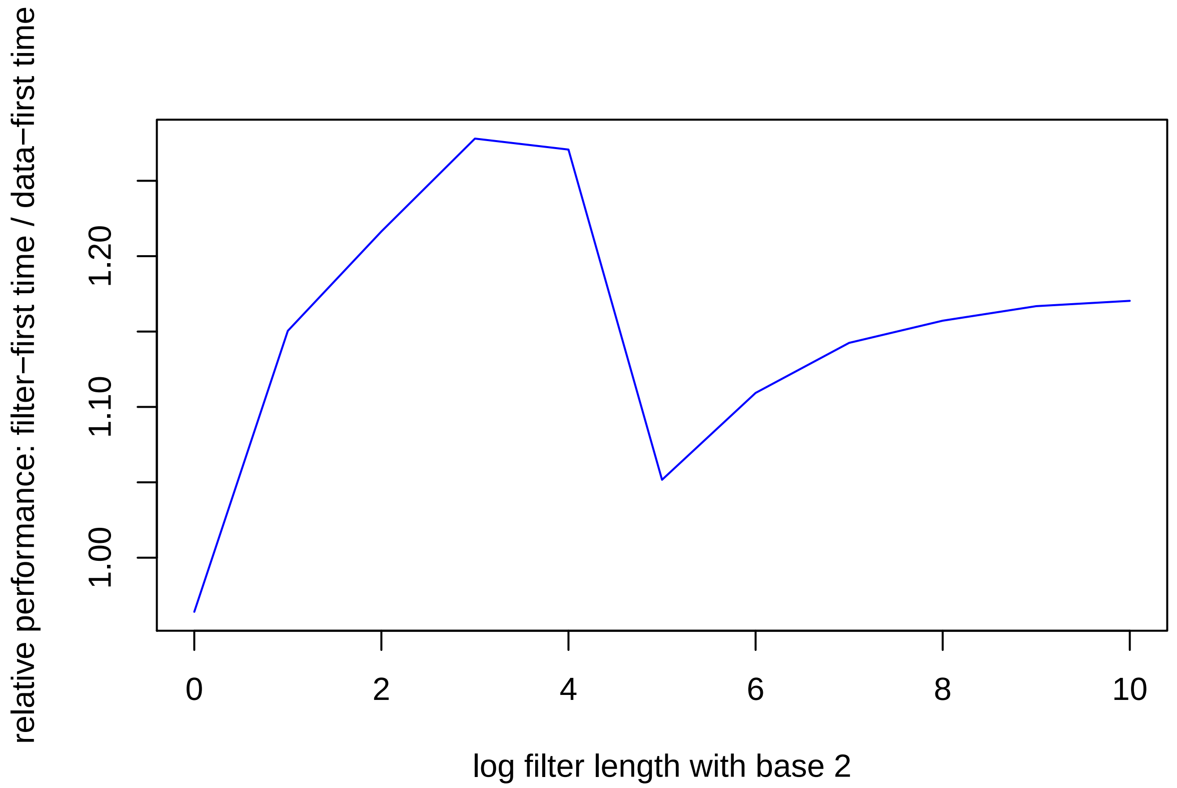


second.

- Is it more efficient to have the data or the filter in the outer loop? Why?

It is more efficient to put the data in the outer loop. Because when the data array is put in the outer loop, in each iteration of the outer loop, the data array stay unchanged, and what's loaded over and over again is the filter array in the inner loop. In our simulation, the filter array is much smaller than the data array, thus to load the filter array frequently is more efficient than to load the data array frequently.

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



The relative performance (filter first running time / data first running time) vs. filter size plot is shown above. When the filter length is 1, the filter-first method is a little faster. For filter length greater than 1, the data-first method performs better. As the filter size increases, the relative performace first oscillates between 1.05 and 1.27 and then approaches to 1.16.

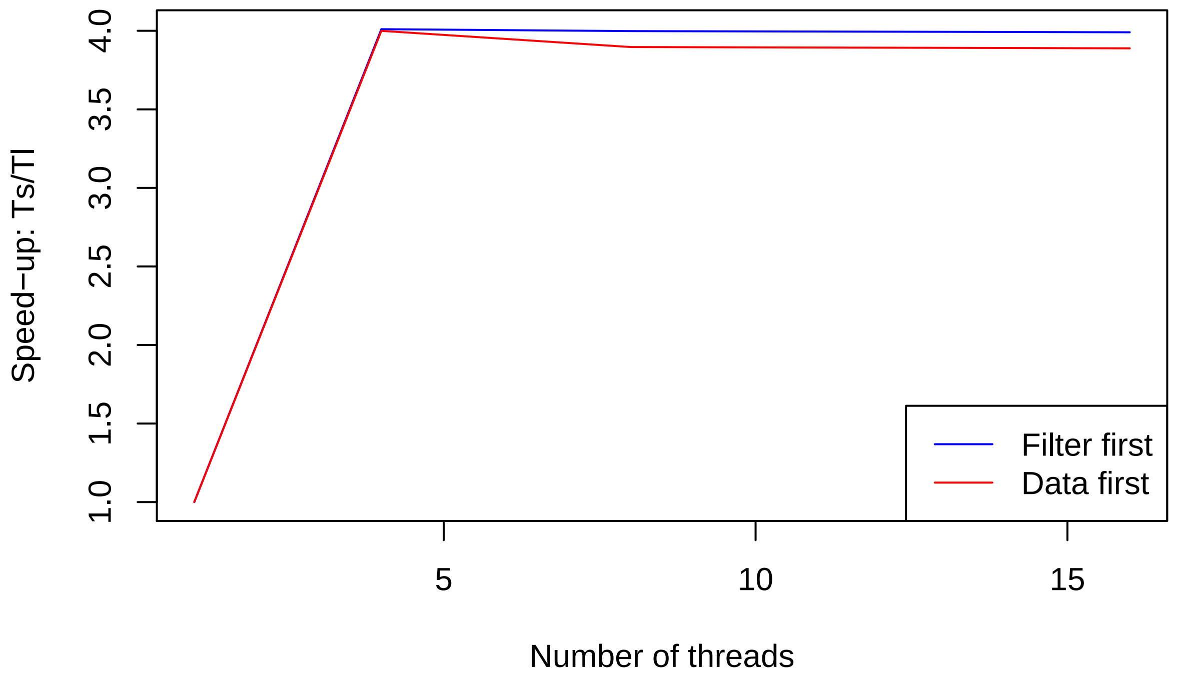
Loop Parallelism

1. (10 pts) Implement loop parallel versions of the code. Run these on filters of length 512.

- Create a function parallelFilterFirst that is the same as serialFilterFirst except that it has the #pragma omp parallel for around the outer loop.
- Create a function parallelDataFirst that is the same as serialDataFirst except that it has the #pragma omp parallel for around the outer loop.

1. (20 pts) Scaleup and speedup

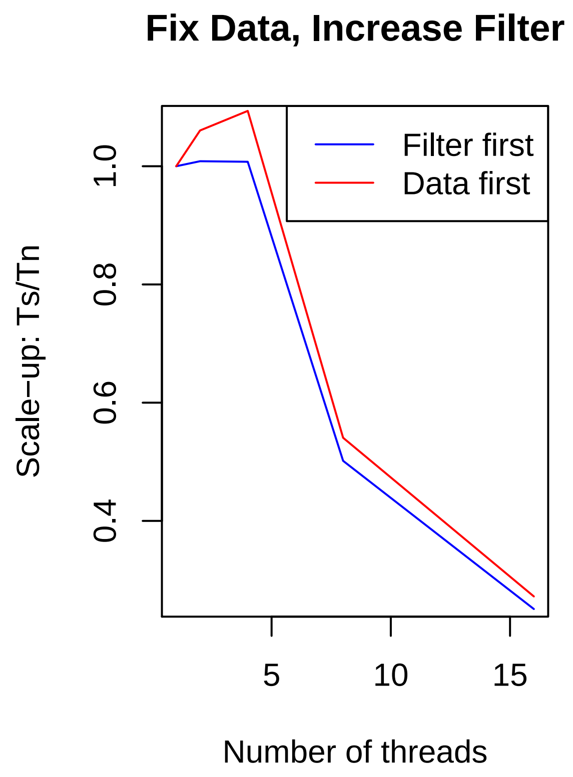
- Generate speedup plots for 1,2,4,8, and 16 threads. You will have to specify the number of threads to run using omp_set_num_threads().

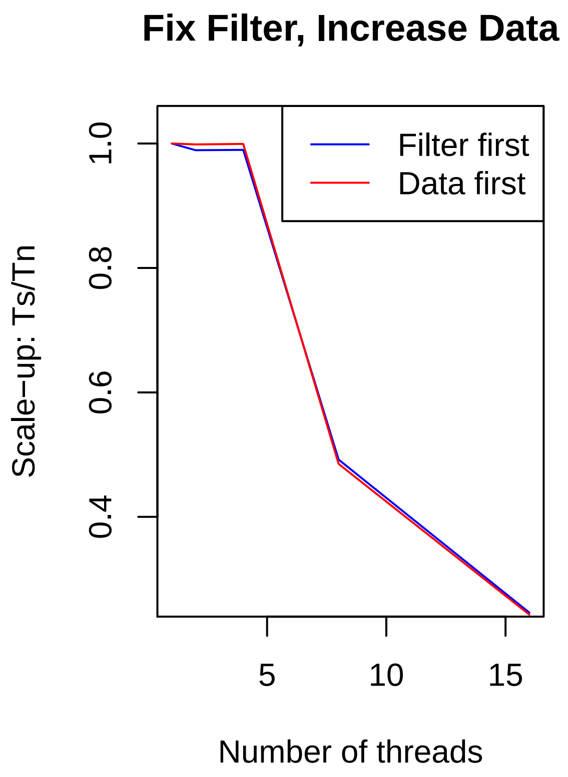


- Describe the results. What type of speedup was realized? When did speedup tail off? Why? What hardware did this run on and how did that influence your result?

The speedup plot suggests that the speedup defined by T_S/T_L increases almost linearly with the number of threads at first and then stops increasing and approaches some constant around 4, which is first achieved with 4 threads. The speedup tails off starting at 4 threads. The experiment was done on a AWS c3.2xlarge instance with 4 physical CPUs. When the number of threads is equal to the number of cores, the best speedup should be achieved because further increasing the number of threads will make threads waiting.

- Generate scaleup plots, by increasing the input size as you increase the parallelism.





- Describe the results. What type of scaleup was realized? When did scaleup tail off? Why? What hardware did this run on and how did that influence your result?

Two types of experiments were done. The left plot shows the case when data length is fixed at (512x512x128) and the filter length is set to 32, 64, 128, 256, 512 as the number of threads is set to 1, 2, 4, 8, 16. The right plot shows the case when filter length is fixed at 512, and the data length is set to (512x512x128)x(1/16, 1/8, 1/4, 1/2,1).

In both cases, sublinear scaleup was realized. Scaleup tail off starting at 4 threads. The experiment was done on a AWS c3.2xlarge instance with 4 physical CPUs. When the number of threads is equal to the number of cores, the best performance should be achieved. Further increasing the number of threads (to 8, 16 or higher) will make threads waiting, and the running time increases almost linearly with data size.

2. (20 pts) Parallel performance.

- Which version of the code was faster? Why?

The data first version is still a little faster. It is the same reason as in the serial case. Because the parallelization is applied to the outer loop, then in each thread, the code running is still serial. However, the advantage over the filter first version is not as significant because there is less computation in each thread than in the seral version. The time saving in each iteration is less accumulated.

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

By Amdahl's law, The overall speedup is

$$k = \frac{1}{(1-p) + \frac{p}{s}}$$

, then we have

$$p = \frac{1/k - 1}{1/s - 1}$$

Since we have 4 physical cores, $s = 4$, and the speedup at $s = 4$ is $k = 3.8966$, thus $p = 0.9912$.

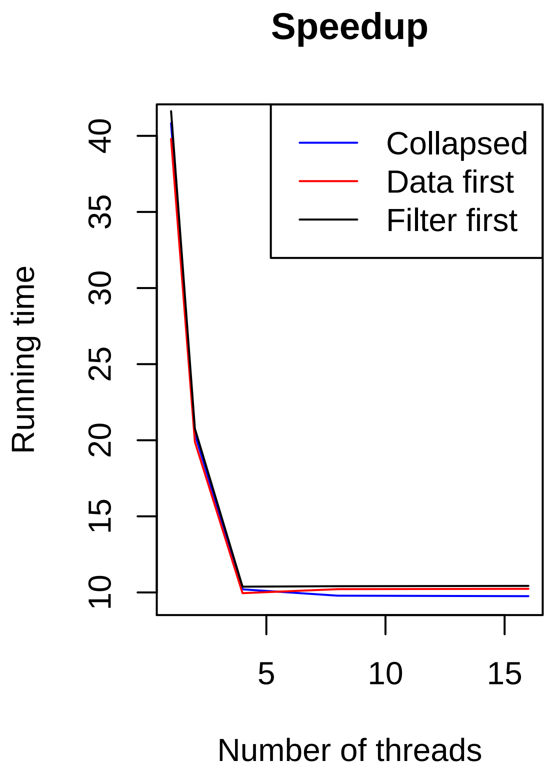
- To what do you attribute any non-ideal speedup or scaleup? To startup costs, interference, skew? Explain your answer.

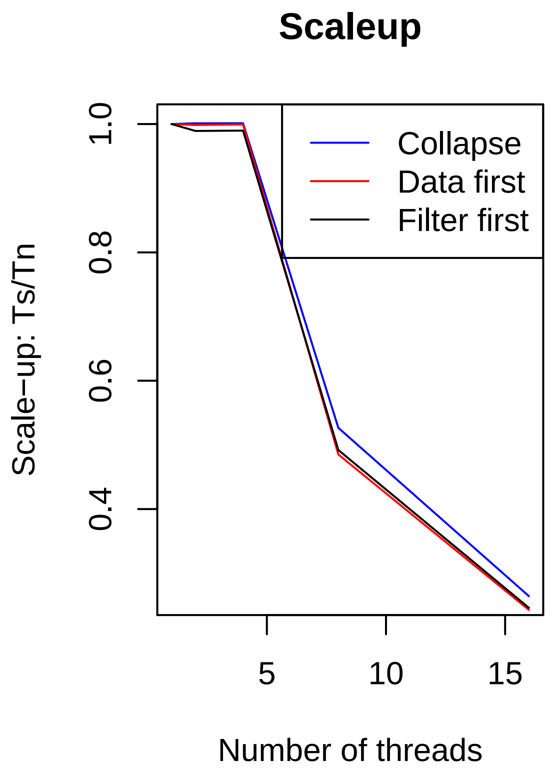
For threads number 1,2,4, the speedup and scaleup are almost linear but still are still not perfect. Such non ideal speedup and scaleup should be attributed to the overheads introduced by coordinating different threads. In addition, when number of threads becomes 8 and 16, hyperthreading is the main issue causing non-ideal speedup and scaleup.

An Optimized Version

1. (30 pts) Optimize the parallel code using techniques described by http://www.akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf. Again run this on a 512 length filter. This is the most important part of the assignment. Please dig deeply.

- Describe your best implementation. What optimizations does it use? Why are they effective?





My best implementation was to put `#pragma omp parallel for collapse(2)` before the outer loop. This method achieves about 5% better performance by collapsing the two nested loops into a large one, and doing parallelization on the collapsed loop. This method is effective because given there is only one loop both the parallelism and the work that each parallelized thread does is simpler and more efficient.

- What other optimizations did you try? Why did they not work or why did you not include them in your best implementation?

Using the same idea, I tried to explicitly write one for loop with calculated the indices for filter and data array, then parallelize this single for loop. Different methods for calculating the indices were tried. But These implements were not included as my best implement because they actally ran longer than the Data-first method. The reason why they were slower is that the overheads caused by calculating the indices of the new loop. In fact, my explicitly written codes were not so efficient as using `#pragma omp parallel for collapse(2)`.