COMP529: Parallel Programming (HW1)

Mohammad Laghari, Najeeb Ahmad

{mlaghari16,nahmad16}@ku.edu.tr

Completed Parts

- Part 1: Parallel Segmentation
- Part 2: Parallel Coloring
- Part 3: Optimization
- Part 4: TBB

1. Report breakdown

The following report is divided as follows. The first section defines the parallel implementation of the program in OpenMP and TBB. Section 2 talks about the optimization techniques. Section 3 discusses the results obtained. And Section 4 shows the results obtained in tabular and graphical format.

2. Parallel Implementation

Parallel implementation was achieved using OpenMP and TBB.

2.1. OpenMP

2.1.1. Image Segmentation

We start by parallelizing the image segmentation function. After careful study of the working of the segmentation, we arrived to the decision of applying parallelization to Loop Nest 1 rather than the iteration of "phases". The loop running on "phases" cannot be parallelized because it must run in a sequential manner for the correct execution of the algorithm. Initially, we used a separate data structure to make changes to the labels which was local to each thread. The idea of using a separate data structure local for each thread was that each thread could make changes to its local copy and then make changes to the global one on its bounded area. However, due to the strategy we devised, the approach of using a separate data structure did not yield accurate results. To avoid this problem, we worked on the global data structure, namely labels. While making a separate data structure would have provided us with a lower completion time and would have removed the underlying data dependencies in loop nest 1 in the "if" condition, using the global array allowed us to make changes in-place and saved us the extra time of allocating and deallocating space for the separate array. Also, the speedup gains of using a separate data structure weren't significant in our implementation of image segmentation. Our implementation divides the image into different segments according to the number of available threads. One might argue that this division would not be fair if the height of the image is odd and the number of available threads is even. To avoid such situations, we divide the image into threads evenly, and if the image cannot be divided evenly, the last thread

handles the remaining strip of the image.
After the labels are assigned to all stripes, the program then runs the second loop nest. We added a line in this loops which checks whether the image has converged. This convergence check will be run on each iteration of "phases" loop. Since the image more often than not converges before the complete iterations of "phases" loop, it is redundant work to run it again. To avoid this situation, our program exits from the image segmentation part if it has converged. This way extra iterations are not run, and this avoids extra time which would otherwise be added in the completion time of segmentation part. The logic behind convergence check is fairly intuitive. A flag marks the value of an integer variable as 1 if the labels did not change in the current iteration. And then if the "phases" loop encounters the variable with value 1, it breaks out.

2.1.2. Coloring

We performed coloring by changing the underlying data structures provided to us for the RGB colors and the count of the clusters. By changing the aforementioned data structures from unordered_maps to arrays, we made sure that the parallel directives could be easily applied to the serial code without introducing any race conditions. Naturally, we started off by using unordered_map by adding parallel directives but encountered race conditions and runtime errors while counting the number of labels and allocating color values to the labels. We identified that the working of unordered_map wasn't supporting parallel directives and races were inhibiting the accuracy. We also identified that using simple arrays would solve this problem with the introduction of only one "atomic" statement while incrementing the count of each label, as atomic clauses are well suited for increment operations. Changing the data structures increased the size of the code as extra conditions had to be added to check for certain color and label values (eg. checks are added to make sure that background colors are not updated or incremented repeatedly). However, the efficiency gains outweigh the effort of computing the few extra if conditions. Color assignment is done by assigning a random color to each label with a count greater than zero.

2.2. Thread Building Block (TBB)

2.2.1. Image Segmentation

The image segmentation with TBB uses the same logic of the image segmentation done in OpenMP. A class defines the loop operator and parallel_for is applied to the height and width for loops from the serial image segmentation code. To maximize the efficiency, we used a 2D parallel_for, i.e., the task are divided on both the height and width of the image. There was no need to break the image into stripes while using TBB since the functioning of TBB and the way it maps threads to tasks handles it on its own.

2.2.2. Coloring

The implementation of coloring in TBB was again fairly intuitive after the parallelization was done in OpenMP. We simply replaced the OpenMP parallel directives with the lambda function calls of parallel_for in TBB. There was one added functionality change in TBB.

The data structure that holds the count of labels was changed to "atomic" to ensure that counting was done atomically. It's merely the pragma omp atomic variant of TBB.

3. Optimizations

We were able to achieve 2 optimizations, Covergence when early termination and changing the data structures. We were, however, unable to optimize the min and max functions

3.1. Convergence and early termination

We successfully achieved the first optimization. We implemented a simple mechanism that checks if the image has converged to it's final form. It does so by checking the values in the labels array. If the label values remain the same in the current iteration, it means that the image has converged to it's final form. A flag is toggled when such a situation (convergence) is encountered. Then after loop nest 1 and loop nest 2, a simple "if" condition checks whether the flag is set to 1. If so, the program breaks out of the image segmentation function and saves time by not redoing the remaining iterations of the "phases" loop.

3.2. Data Structures

The second optimization we achieved was of finding better data structures in coloring. Although the ideal choice for a serial program would have been an unordered_map, in the parallel version of the program, the traversal of this data structure was becoming a bottleneck to the successful execution of the program. We identified that these bottlenecks could be removed by replacing unordered_map with arrays. Arrays allowed us to traverse through our items in parallel and perform desired operations.

4. Results

We ran our experiments on lufer with the excl flag set true to achieve fair results for benchmarking purposes.

4.1. OpenMP

Our results include tests for parallel implementation using 1 thread only. The results with 1 thread are kept as they have updated data structures. When compared with the serial execution times, gains in terms of total execution times can be seen clearly. Figure 1 shows results for segmentation time in OpenMP for the images coffee.jpg, windows.jpg and hanger.jpg. The image shows that the time for segmentation reduced drastically and a speedup of 6.8 was achieved using 32 threads. Figure 2 shows results for coloring time in OpenMP. A similar result set was obtained with a speedup of 3.17 was achieved using 32 threads. Figure 3 shows the total time taken to complete the experiment, i.e. Segmentation + Coloring time. Here, a speedup of 6.01 was achieved using OpenMP with 32 threads on coffee.jpg. Table 1 shows a detailed view of execution times of the three images and their respective speedups

4.2. Thread Building Blocks (TBB)

Using TBB we achieved a speedup of 5.2 in total time of execution as shown in Figure 6 coffee.jpg. Figure 4 and 5 show the trend of image segmentation and coloring time with a speedup of 5.58 and 3.18 respectively for coffee.jpg. Table 2 also shows the detailed completion times of the three images.

5. Figures and Tables

Tables 1 and 2 show the data for Segmentation Times. Tables 3 and 4 show the data for Coloring Times. And tables 5 and 6 show the data for Total times. Figures 1 to 3 and 4 to 6 show graphical data for Segmentation, Coloring and Total time for OpenMP and TBB respectively.

Table 1: Results obtained after parallelization with OpenMP for Total Time (Segmentation + Coloring).

	Serial	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread
Coffee.jpg	20.46205	12.872729	8.843818	7.36898	4.556663	3.544661	3.4034
Windows.jpg	12.402963	6.02963	6.126953	3.576949	2.337155	1.953199	1.839675
Hanger.jpg	11.652787	6.766386	6.320402	3.735462	2.625809	2.073324	2.022045

Table 2: Results obtained after parallelization with TBB for Total Time (Segmentation + Coloring).

	Serial	2 thread	4 thread	8 thread	16 thread	32 thread
Coffee.jpg	20.46205	11.377808	7.046242	4.304106	3.691193	3.939348
Windows.jpg	12.402963	7.586325	4.289322	2.766463	2.577345	2.490793
Hanger.jpg	11.652787	5.037261	4.075787	2.735846	2.271745	2.322343

Table 3: Results obtained after parallelization with OpenMP for Segmentation Time.

	Serial	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread
Coffee.jpg	18.400991	12.414203	8.327897	6.839323	4.042059	2.985602	2.753985
Windows.jpg	11.45933	5.682682	5.772465	3.281169	2.024092	1.580987	1.458688
Hanger.jpg	10.348809	6.460941	5.986893	3.448923	2.327937	1.757668	1.681007

Table 4: Results obtained after parallelization with TBB for Segmentation Time.

	Serial	2 thread	4 thread	8 thread	16 thread	32 thread
Coffee.jpg	18.400991	10.969491	6.6162	3.787564	3.155187	3.292722
Windows.jpg	11.45933	7.305731	4.013777	2.512576	2.22543	2.141207
Hanger.jpg	10.348809	4.788919	3.859281	2.487749	1.920038	1.998645

Table 5: Results obtained after parallelization with OpenMP for Coloring Time.

	Serial	1 thread	2 thread	4 thread	8 thread	16 thread	32 thread
Coffee.jpg	2.061059	0.458526	0.515921	0.529657	0.514604	0.559059	0.649415
Windows.jpg	0.943633	0.346948	0.354488	0.29578	0.313063	0.372212	0.380987
Hanger.jpg	1.303978	0.305445	0.333509	0.286539	0.297872	0.315656	0.341038

Table 6: Results obtained after parallelization with TBB for Coloring Time.

	Serial	2 thread	4 thread	8 thread	16 thread	32 thread
Coffee.jpg	2.061059	0.408317	0.430042	0.516542	0.536006	0.646626
Windows.jpg	0.943633	0.280594	0.275545	0.253887	0.351915	0.349586
Hanger.jpg	1.303978	0.248342	0.216506	0.248097	0.351707	0.323698

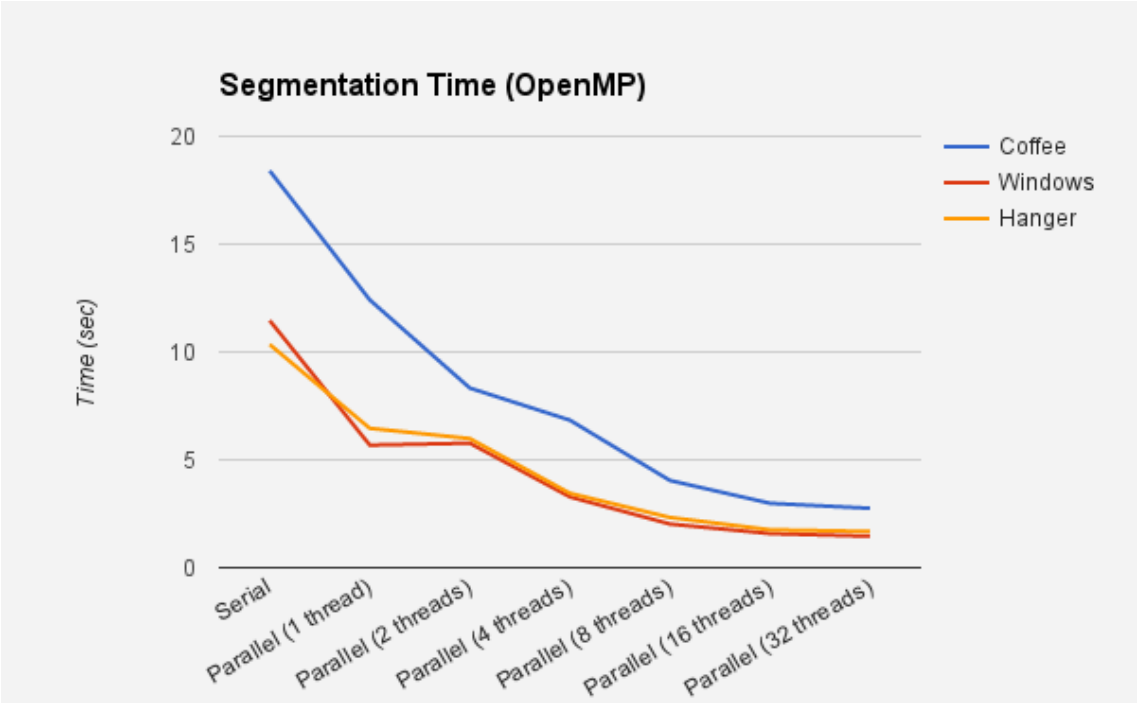

Figure 1

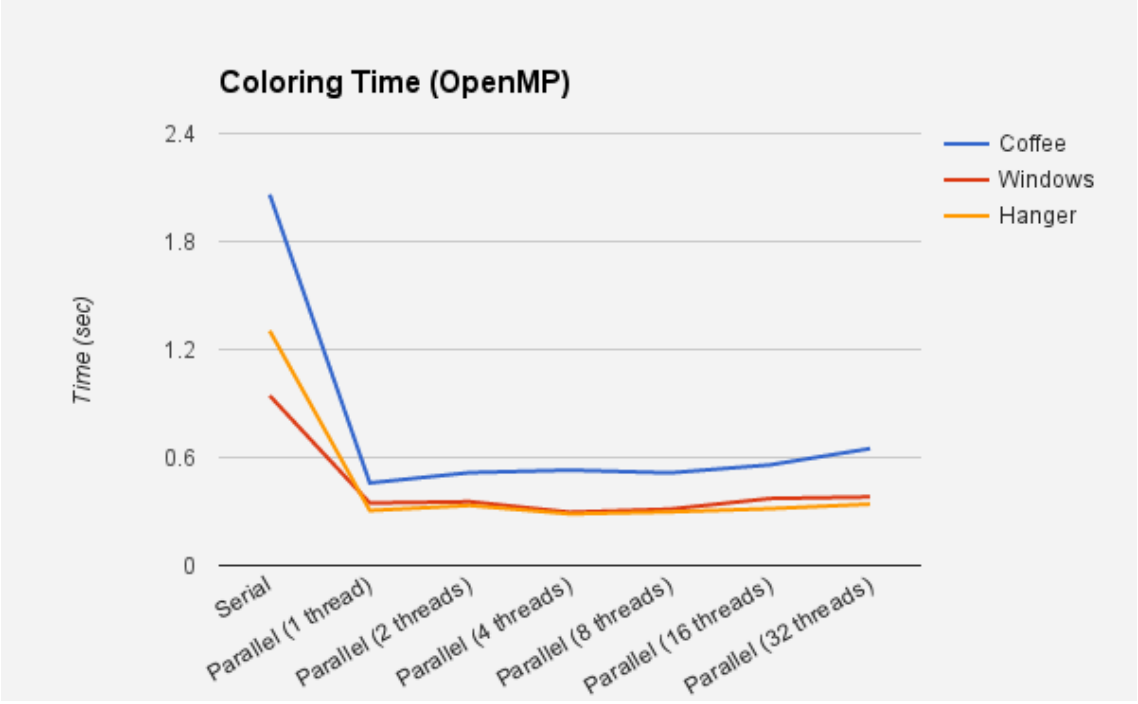

Figure 2

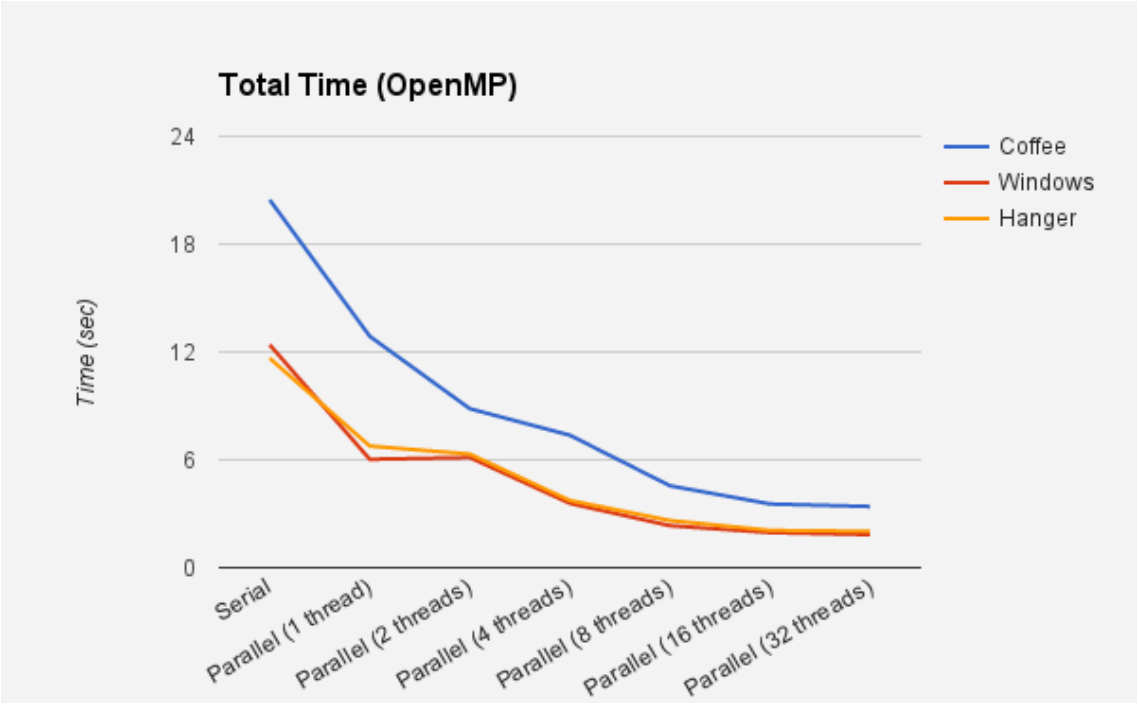

Figure 3

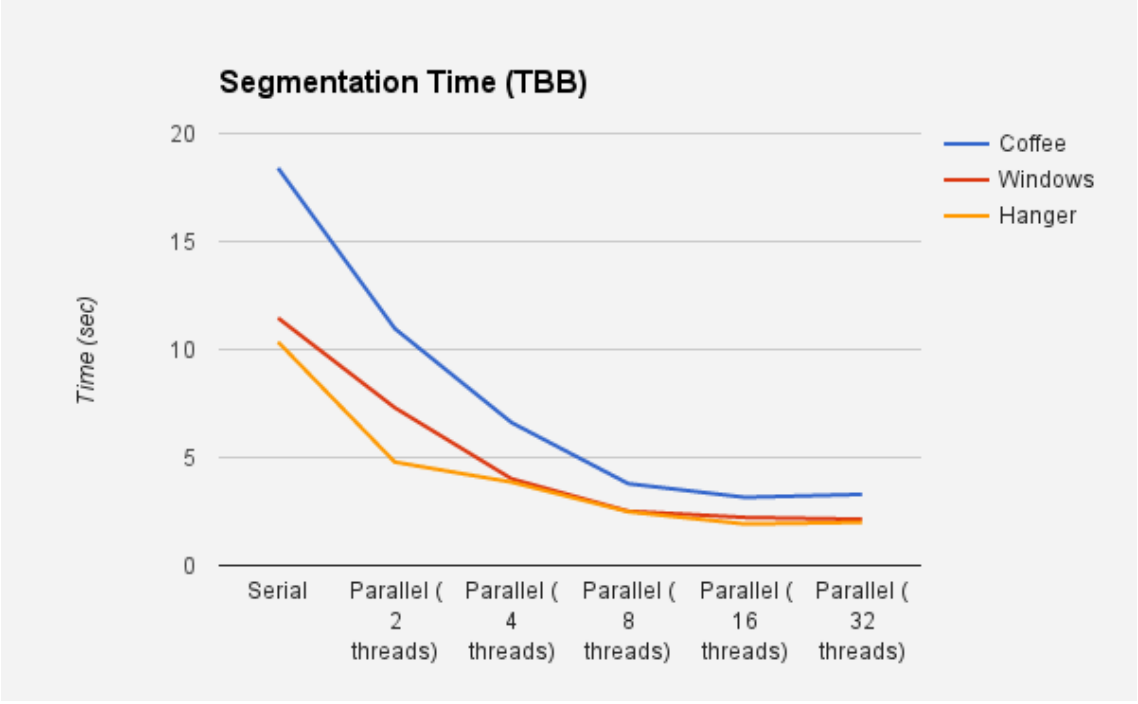

Figure 4

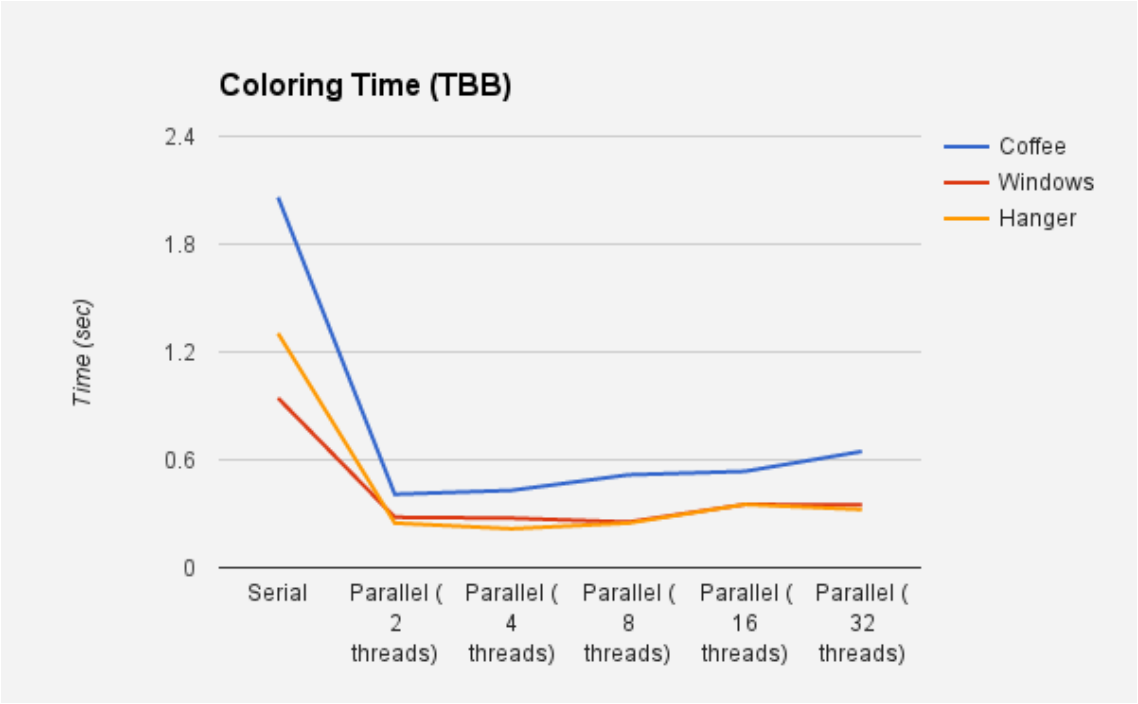

Figure 5

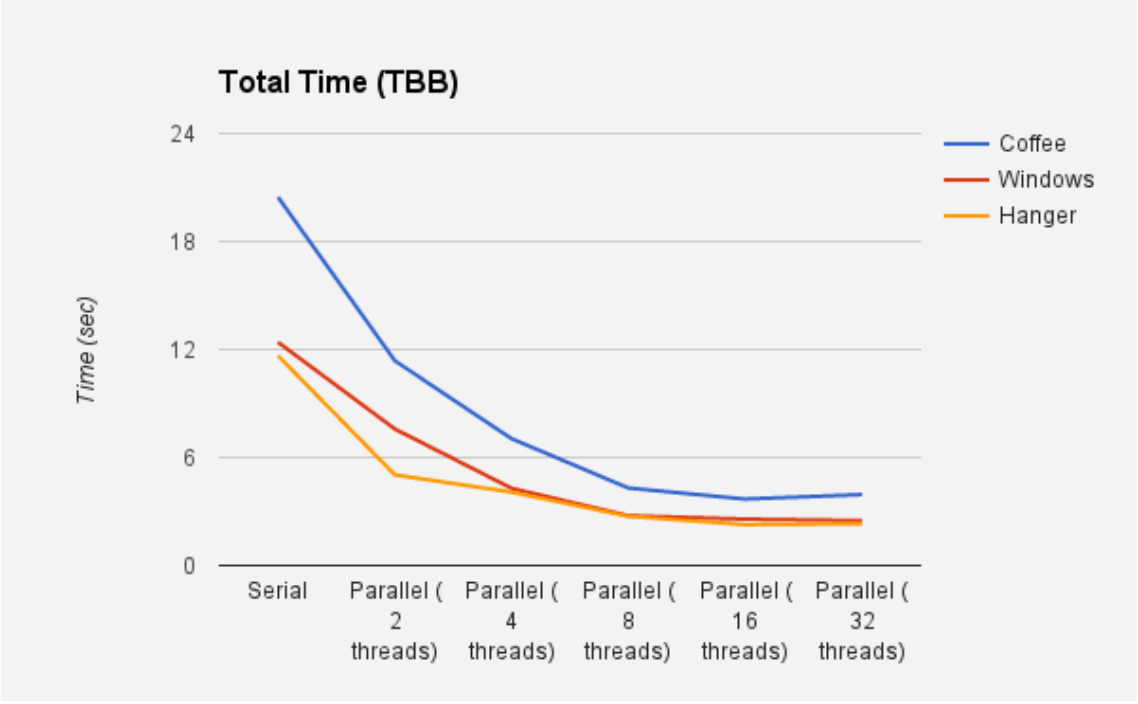

Figure 6