2024

Parallel Digital Image Processing and Analysis

PROJECT REPORT AND ANALYSIS – VERSION 2.0

RAFAEL FAÍSCA 1180658
ÓSCAR FOLHA 1181600

Index

Index of figures

Index of Tables

Document Approval

Name	Date	Signature
Óscar Folha	27/04/2024	Óscar Folha
Rafael Faísca	27/04/2024	Rafael Faísca

Revision history

Version	Author	Description	Date
1.0	Óscar Folha	Initial version, divide the document.	25-04-2024
2.0	Everyone	Final version	28-04-2024

1 Introduction

1.1 Purpose

The purpose of this project is to investigate the performance and efficiency of parallel digital image processing and analysis techniques. By applying various filters to a set of images using different execution methods, the study aims to identify the most effective approach to processing digital images at scale. The key focus will be on evaluating sequential execution, multithreading, and thread pools, as well as examining the impact of different Java garbage collectors on performance. This research will provide insights into how these techniques can be leveraged to optimize image processing tasks, which is increasingly relevant in a world where large volumes of digital images are processed daily for various applications, such as medical imaging, remote sensing, and computer vision.

1.2 Project scope

This project aims to conduct an in-depth analysis of parallel digital image processing and analysis, focusing on applying various filters to a set of images using different Java-based execution strategies. The filters to be applied include brighter, blur, grayscale, swirl, glass, and conditional blur. The project will explore multiple execution paradigms, specifically sequential execution, multithreading, and different configurations of thread pools, such as CompletableFuture, ForkJoin, and ExecutorService-based approaches. Additionally, the study will examine these thread pool methods with and without data partitioning to understand their impact on performance.

The scope also encompasses a comprehensive examination of Java garbage collectors to determine their influence on processing efficiency. By assessing these diverse execution strategies and garbage collection methods, the project will provide insights into the most effective techniques for parallel image processing within the context of SISMD or Sistemas Multinúcleo e Distribuídos. This research aims to contribute to optimizing image processing tasks in various applications, offering a valuable guide for practitioners seeking to enhance the performance and scalability of their image processing systems.

2 Images Selected and Filter Algorithms Applied

To ensure a comprehensive analysis for our research, we selected four distinct images with varying dimensions. This diversity in image sizes is crucial to evaluate the performance and robustness of our application across different scenarios.

The images used in this study are as follows:

696 x 522, turtle.jpg: A smaller image, ideal for testing basic processing speed and memory usage.

Figure 1 - turtle.jpg

1920 x 1080, planets.jpg: A widely used high-definition format, representing standard HD resolution.

Figure 2 - planets.jpg

5472 x 3648, paisagem.jpg: A large image, suitable for assessing how the application handles larger data sets.

Figure 3- paisagem.jpg

7680 x 4320, sunset.jpg: An ultra-high-definition image, pushing the boundaries of performance and resource allocation.

Figure 4 - sunset.jpg

By including images of varying dimensions, we can effectively determine the ability of our application to maintain good performance while processing diverse workloads. This approach allows us to simulate real-world conditions and ensure that our results are relevant and applicable to a broad range of image processing tasks.

2.1 Brighter

This filter increases the brightness of an image by adjusting the intensity of its pixels. It's useful for enhancing images that are too dark, making details more visible.

Figure 5 - turtle.jpg brighter

Figure 6- planets.jpg brighter

Figure 7- paisagem.jpg brighter

Figure 8- sunset.jpg brighter

2.2 Grayscale

This filter converts a color image into grayscale by removing all color information, resulting in shades of gray from black to white. It's often used for artistic effects or simplifying image data.

Figure 9 - turtle.jpg grayscale

Figure 10 - planets.jpg grayscale

Figure 11- paisagem.jpg grayscale

Figure 12- sunset.jpg grayscale

2.3 Swirl

The swirl filter creates a spiral effect in the image, distorting its elements in a circular pattern. This filter is used for creative effects or for generating unique transformations.

Figure 13- turtle.jpg swirl

Figure 14- planets.jpg swirl

Figure 15- paisagem.jpg swirl

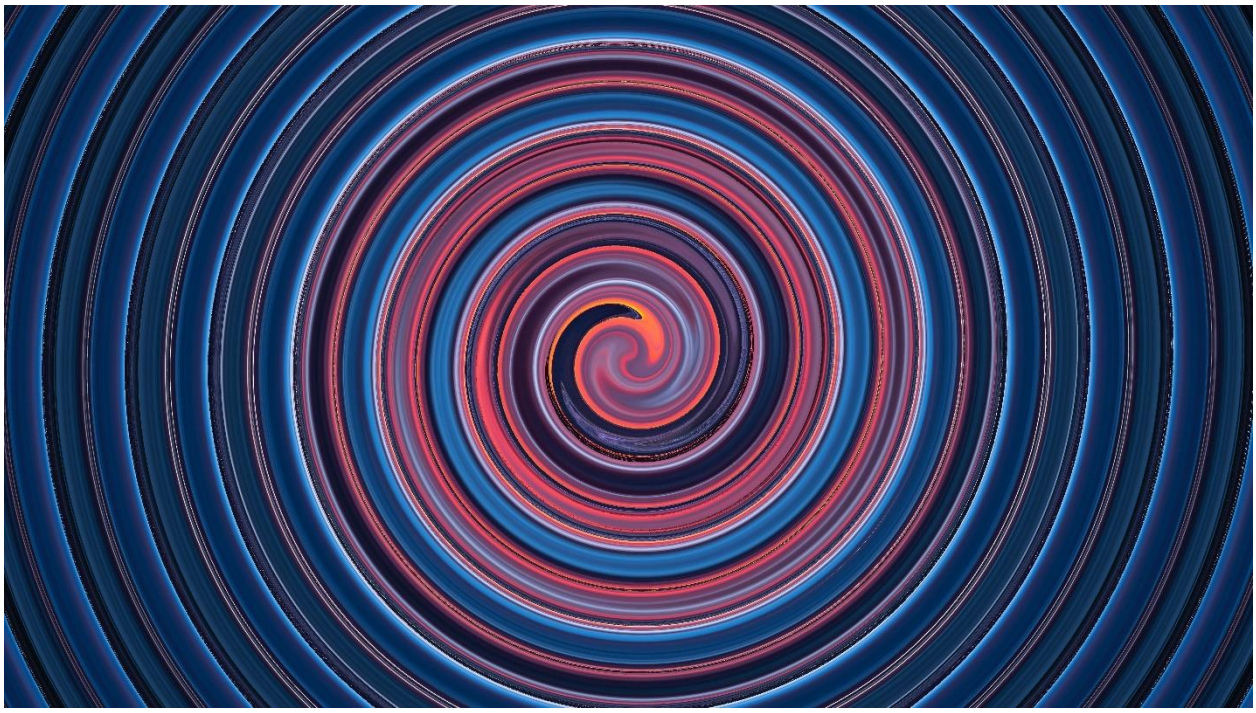

Figure 16 - sunset.jpg swirl

2.4 Glass

This filter simulates the appearance of viewing an image through a pane of glass with distortions. It creates a unique, disjointed effect that can be used for artistic or surreal purposes.

Figure 17- turtle.jpg glass

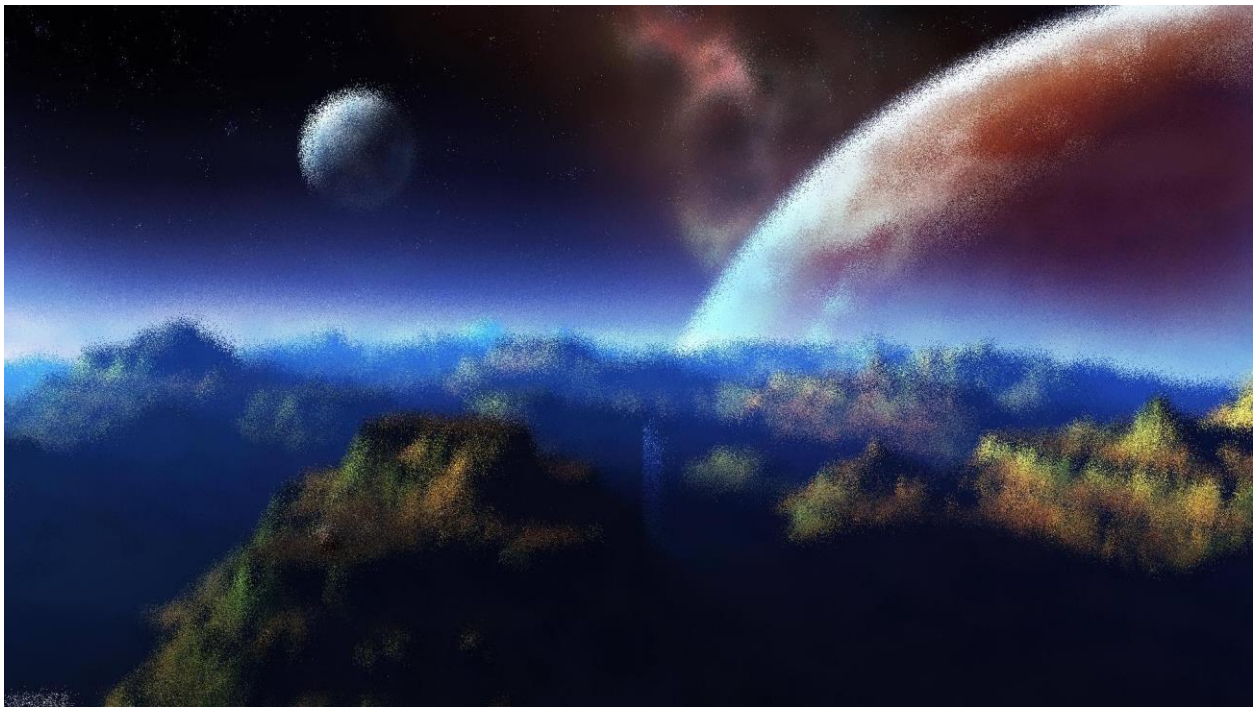

Figure 18- planets.jpg glass

Figure 19- paisagem.jpg glass

Figure 20- sunset.jpg glass

2.5 Blur

The blur filter smooths out details in an image, reducing noise and making it less sharp. It's commonly used for creating a softened effect or for background defocusing.

Figure 21- turtle.jpg blur

Figure 22- planets.jpg blur

Figure 23- paisagem.jpg blur

Figure 24- sunset.jpg blur

2.6 Conditional Blur

This filter applies a blur effect only to specific regions or elements within an image based on defined conditions. It's useful for emphasizing certain areas while de-emphasizing others, offering a flexible approach to image processing.

Hard to spot the difference between the original but the white spots are a little darker and distorted.

Figure 25- turtle.jpg conditional blur

Figure 26- planets.jpg conditional blur

Figure 27- paisagem.jpg conditional blur

Figure 28- sunset.jpg conditional blur

3 How to run the application and use different Garbage Collectors

To run our application, we recommend using the JAR file located in the StarterCode folder within the deliverable ZIP archive. The file is named StarterCodeExec.jar. Once you've located it, ensure that you create the following folders in the directory where you intend to run the JAR file:

- sequential
- multithread
- threadpoolexecutor
- threadpoolexecutorpartition
- threadpoolforkjoin
- threadpoolcompletablepartitions

These folders will be used to store the generated images from different processing methods.

To select a specific garbage collector (GC), use one of the following commands before running the JAR file:

For Parallel GC:

java -XX:+UseParallelGC -jar .\StarterCodeExec.jar

For G1 GC:

java -XX:+UseG1GC -jar .\StarterCodeExec.jar or no command needed since G1 is Java's default.

For Shenandoah GC:

java -XX:+UseShenandoahGC -jar .\StarterCodeExec.jar

For Z GC:

java -XX:+UseZGC -jar .\StarterCodeExec.jar

Select the desired GC option based on your performance and memory management requirements. This setup allows you to evaluate the application under different garbage collection strategies, providing flexibility for testing and benchmarking.

You can also add environment variables into the java command.

For exemple:

Java -DNPARTITIONS=30 -jar .\StarterCodeExec.jar

- **BRIGHTER_VALUE**: Defines the brightness level. It defaults to 128 but can be adjusted by setting the BRIGHTER_VALUE system property.
- **GLASS_MAXIMUM_DISTANCE**: Represents the maximum distance for a glass effect, with a default value of 10. It can be configured by setting the GLASS_MAXIMUM_DISTANCE system property.
- **BLUR_SUBMATRIX** and **CONDITIONALBLUR_SUBMATRIX**: Control the size of the blur matrix, with both defaulting to 5. They can be changed by setting BLUR_SUBMATRIX and CONDITIONALBLUR_SUBMATRIX.
- **RED_TRESHOLD, BLUE_TRESHOLD, and GREEN_TRESHOLD**: Set the thresholds for the red, blue, and green color channels, with defaults of 100, 150, and 100, respectively. You can adjust these by setting RED_TRESHOLD, BLUE_TRESHOLD, and GREEN_TRESHOLD.
- **NPARTITIONS**: Specifies the number of partitions, defaulting to 16. This can be changed with the NPARTITIONS system property.
- **THRESHOLD_RECURSIVE**: Represents a threshold value for recursive operations, with a default of 100. You can set it via the THRESHOLD_RECURSIVE system property.

4 Results/Analysis

To generate the results discussed in this section, we applied the following configurations: The brighter filter was set to a value of 126, while the glass filter had a maximum distance of 10. The blur filter used a submatrix size of 5, and the conditional blur filter also utilized a submatrix size of 5, with specific values that may not be optimal for all images. For the executor-based and CompletableFuture approaches, the partition count was 16, and the recursive threshold for the ForkJoin framework was 100.

Our approach involved presenting the results for each type of implementation on two separate computers, analyzing the performance on each, and then drawing conclusions based on the results from both machines. This structure allows for a detailed examination of how the different configurations and hardware environments impact the performance

and outcomes of the image processing tasks. We will first present the results for each implementation type, then proceed with an analysis of each computer's performance, and conclude with a summary comparing the outcomes from both systems.

FYI: In the deliverable ZIP there's a excel file that contains all the results that will be shown in this chapter. All the results here are presented in ms.

4.1 Óscar's PC

Óscar computer's specs are: 8 logic processors, processor 11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz 3.30 GHz, ram 16,0 GB.

The jdk used on this computer wasn't compatible with **Shenandoah GC**.

4.1.1 Sequential

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	14	22	19	31	101	7	194
1920 x 1080	190	77	48	95	637	73	1120
5472 x 3648	1240	323	416	987	6427	177	9570
7680 x 4320	2795	1695	716	1679	10649	728	18262

Table 1 - Parallel GC Sequential

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	21	20	22	32	108	9	212
1920 x 1080	79	56	57	100	565	90	947
5472 x 3648	349	194	409	1010	5743	157	7862
7680 x 4320	975	722	738	1657	9526	833	14451

Table 2 - G1 GC Sequential

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	15	15	29	39	146	10	254
1920 x 1080	53	55	56	105	626	75	970
5472 x 3648	568	559	492	1572	7215	277	10683
7680 x 4320	1309	2352	949	2662	13396	2238	22906

Table 3 - Z GC Sequential

4.1.2 Multithread

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	16	17	17	22	34	13	119
1920 x 1080	31	31	27	41	135	73	338
5472 x 3648	262	201	101	242	2239	64	3109
7680 x 4320	528	938	144	425	4027	309	6371

Table 4 - Parallel GC Multithread

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	18	24	19	23	33	10	127
1920 x 1080	34	29	28	44	142	52	329
5472 x 3648	113	262	106	258	2074	58	2871
7680 x 4320	1211	512	358	710	3428	325	6544

Table 5 - G1 GC Multithread

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	20	20	17	26	28	10	121
1920 x 1080	42	36	29	56	202	56	421
5472 x 3648	187	509	217	401	5847	120	7281
7680 x 4320	363	2020	236	934	11272	714	15539

Table 6 - Z GC Multithread

Threadpool Executor Based

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	23	6	16	13	26	6	90
1920 x 1080	45	36	25	33	174	36	349
5472 x 3648	180	285	75	241	1756	42	2579
7680 x 4320	1430	993	131	422	4209	304	7489

Table 7 - Parallel GC Threadpool Executor Based

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	21	6	20	15	33	5	100
1920 x 1080	34	34	28	37	159	38	330
5472 x 3648	123	121	93	242	2143	57	2779
7680 x 4320	326	458	281	454	3515	389	5423

Table 8 - G1 GC Threadpool Executor Based

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	26	22	18	19	36	6	127
1920 x 1080	22	21	24	22	198	63	350
5472 x 3648	191	442	160	354	5010	113	6270
7680 x 4320	518	1441	277	723	5008	1717	9684

Table 9 - Z GC Threadpool Executor Based

4.1.3 Threadpool Executor Based with 16 Partitions

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	13	3	2	4	23	3	48
1920 x 1080	13	12	18	20	131	19	213
5472 x 3648	156	672	84	215	1675	43	2845
7680 x 4320	1490	1123	128	365	4027	223	7356

Table 10 - Parallel GC Threadpool Executor Based with 16 Partitions

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	3	3	15	4	34	3	62
1920 x 1080	29	13	9	20	162	25	258
5472 x 3648	100	236	91	225	2082	38	2772
7680 x 4320	165	1058	274	399	3502	189	5587

Table 11 - G1 GC Threadpool Executor Based with 16 Partitions

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	5	7	2	5	37	5	61
1920 x 1080	35	38	9	21	159	25	287
5472 x 3648	171	549	75	363	5016	77	6251
7680 x 4320	315	1325	197	748	5010	472	8067

Table 12 - Z GC Threadpool Executor Based with 16 Partitions

4.1.4 Threadpool Fork Join Based (100 threshold)

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	11	3	11	5	21	5	56
1920 x 1080	16	17	9	23	131	20	216
5472 x 3648	153	740	85	220	1700	44	2942
7680 x 4320	1221	574	123	375	4263	199	6755

Table 13 - Parallel GC Threadpool Fork Join Based (100 threshold)

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	4	3	4	5	34	5	55
1920 x 1080	13	24	10	24	181	20	272
5472 x 3648	100	99	81	225	1972	40	2517
7680 x 4320	219	909	240	411	3534	168	5481

Table 14 - G1 GC Threadpool Fork Join Based (100 threshold)

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	4	5	4	4	30	5	52
1920 x 1080	20	18	11	25	151	30	255
5472 x 3648	174	513	81	376	5983	71	7198
7680 x 4320	387	1444	198	750	11809	342	14930

Table 15 - Z GC Threadpool Fork Join Based (100 threshold)

4.1.5 Threadpool Comparable Future Based 16 Partitions

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	24	3	3	14	41	12	97
1920 x 1080	20	27	14	35	126	29	251
5472 x 3648	173	271	84	220	1680	48	2476
7680 x 4320	1429	1024	154	378	4492	213	7690

Table 16 - Parallel GC Threadpool Comparable Future Based 16 Partitions

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	8	12	3	12	41	20	96
1920 x 1080	30	28	17	36	148	35	294
5472 x 3648	100	180	86	236	1941	50	2593
7680 x 4320	1146	494	279	689	3493	236	6337

Table 17 - G1 GC Threadpool Comparable Future Based 16 Partitions

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	23	16	12	11	30	12	104
1920 x 1080	27	35	19	30	168	36	315
5472 x 3648	194	841	161	347	5686	123	7352
7680 x 4320	370	1445	216	701	10102	617	13451

Table 18 - Z GC Threadpool Comparable Future Based 16 Partitions

4.1.6 General Results

Total Time per GC

GC	Parallel	G1	Z
Total Time	80535	68299	132929

Table 19 - Total time per GC

Total Time per Executor Method

Executor	Sequential	Multithread	Executor	Executor Partition	ForkJoin	CompletableFuture
Nº of best results	0	1	1	5	4	1
Total Time	87431	43170	35570	33807	40729	41056

Table 20 - Total Time per Executor Method

Total Time Executor Method per Image Size

	Sequential	Multithread	Executor	Executor Partition	ForkJoin	CompletableFuture
696 x 522	660	367	317	171	163	297
1920 x 1080	3037	1088	1029	758	743	860
5472 x 3648	28115	13261	11628	11868	12657	12421
7680 x 4320	55619	28454	22596	21010	27166	27478

Table 21 - Total Time of each method per Image Size

4.1.7 Partitions and Thresholds Variation

4.1.7.1 Partition Variation with the better GC

8 Partitions

	8 partitions – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	30	22	24	19	60	10	165
1920 x 1080	79	31	31	32	205	47	425
5472 x 3648	313	144	109	267	2029	70	2932
7680 x 4320	863	572	165	706	3552	279	6137

Table 22 - 8 Partitions with G1 GC

16 Partitions

	16 partitions – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	3	3	15	4	34	3	62
1920 x 1080	29	13	9	20	162	25	258
5472 x 3648	100	236	91	225	2082	38	2772
7680 x 4320	165	1058	274	399	3502	189	5587

Table 23 - 16 Partitions with G1 GC

32 Partitions

	32 partitions – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	27	21	21	21	56	9	155
1920 x 1080	90	49	49	56	227	44	515
5472 x 3648	310	137	123	421	2150	60	3201
7680 x 4320	662	546	216	424	3542	288	5678

Table 24 - 32 Partitions with G1 GC

Results

Partitions	8	16	32
Total	9659	8679	9549
	3º	1º	2º

Table 25 - Final results of partition tests

4.1.7.2 Threshold Variation with the better GC

Threshold 50

	Thershold 50 – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	4	10	21	17	35	5	92
1920 x 1080	19	16	31	22	191	21	300
5472 x 3648	243	223	94	1960	42	40	2602
7680 x 4320	1192	495	143	399	3508	291	6028

Table 26 - Threshold 50 with G1 GC

Threshold 100

	Thershold 100 – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	4	3	4	5	34	5	55
1920 x 1080	13	24	10	24	181	20	272
5472 x 3648	100	99	81	225	1972	40	2517
7680 x 4320	219	909	240	411	3534	168	5481

Table 27 - Threshold 100 with G1 GC

Threshold 150

	Thershold 150 – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	4	11	16	16	32	7	86
1920 x 1080	31	31	32	33	210	31	368
5472 x 3648	110	243	95	235	1968	47	2698
7680 x 4320	477	1145	141	455	3576	310	6104

Table 28 - Threshold 150 with G1 GC

Results

Threshold	150	100	50
Total	9256	8325	9022
	3º	1º	2º

Table 29 - Final results of the Threshold tests

4.1.8 Conclusions

4.1.8.1 Garbage Collector Performance

Total Times Across Different Garbage Collectors (GC)

Parallel GC: Performs well with larger photo dimensions, but its total time is higher compared to G1. This suggests it might have more overhead or less efficient memory management.

G1 GC: Generally consistent across different dimensions and execution methods. Its balanced performance makes it a strong choice for high-throughput applications.

Z GC: Shows the longest total time, indicating it may not be ideal for workloads requiring high performance. Z GC focuses on ultra-low latency but might not be optimized for high throughput.

4.1.8.2 Execution Methods Comparison

4.1.8.2.1 Sequential Execution

Sequential processing is consistently slower compared to other methods, indicating that threading or parallel execution provides a significant speedup. It's not suitable for applications requiring high performance.

4.1.8.2.2 Multithreaded Execution

Multithreaded execution is faster than sequential, but not the fastest overall. It improves performance, but there's potential for optimization through better task distribution and management.

4.1.8.2.3 Executor-Based Threadpool

Without Partitions: Generally, performs well, especially on larger photo dimensions. It leverages multiple threads but might not optimize task distribution as effectively as partitioned approaches.

With Partitions: Shows better performance compared to the unpartitioned approach, particularly on larger photo dimensions. Partitioning allows for more efficient task distribution, reducing overhead and improving performance.

Difference Between with and Without Partitions

Partitioning allows tasks to be divided into smaller, manageable units, enabling more efficient use of multiple threads. This leads to reduced overhead and improved performance compared to non-partitioned methods.

4.1.8.2.4 Fork-Join

Performs well, especially on smaller photo dimensions. The dynamic partitioning nature of Fork-Join allows it to break down tasks and allocate them as needed, optimizing parallel execution.

4.1.8.2.5 Completable Future with Partitions

This method tends to perform great across all photo dimensions, demonstrating the efficiency of using partitioning with completable futures for asynchronous task management. This approach can dynamically distribute tasks and manage dependencies effectively. It still falls behind fork-join and executor with partitions.

4.1.9 Threshold and Partition Variations

4.1.9.1 Threshold Variation

Testing with different thresholds (50, 100, 150) using G1 GC revealed that Threshold 100 is the most consistent across all tests. It provides a balanced compromise between processing time and resource management, yielding optimal performance.

4.1.9.2 Partition Variations

16 Partitions: Achieves the lowest total time (8,679 ms), indicating it's the most efficient configuration among the tested partition counts. It strikes a good balance between parallelism and thread management overhead.

8 Partitions: Has the highest total time (9,659 ms). While fewer partitions can reduce overhead, it might limit the advantages of parallelism, leading to increased total times.

32 Partitions: Has a lower total time than 8 partitions but is still higher than 16 partitions (9,549 ms). This could be due to increased overhead from managing more partitions, offsetting gains from additional parallelism.

Thus, in the context of running filters on a photo, 16 partitions seem to offer the best balance between parallelism and management overhead. This suggests that partitioning plays a significant role in performance, with a moderate number of partitions yielding better results.

4.1.10 Analysis and Expectations of The Best Results

Partitioning and Executor-Based Methods

Partitioning allows tasks to be spread across multiple threads efficiently, reducing idle time and allowing better use of CPU cores. Executor-based methods manage thread allocation, contributing to better performance when combined with partitioning.

Fork-Join

Fork-Join's dynamic partitioning capability helps optimize performance, especially for tasks with variable execution times. This method can adapt to the workload's demands, enhancing efficiency.

Garbage Collectors

G1 GC's consistent performance can be attributed to its efficient memory management across larger workloads, whereas Parallel GC may have more overhead due to its design. This could lead to reduced performance when handling large workloads.

4.1.11 Conclusion

Based on the analysis, the most efficient method for running filters on photos is **Executor-Based with Partitions**. This method outperformed other execution approaches, offering the best total time across various photo dimensions. Its efficiency stems from partitioning tasks into smaller units and executing them concurrently, which maximizes the use of system resources while minimizing overhead.

Fork-Join is a strong contender, performing best with smaller photo dimensions, due to its dynamic task division. It remains a good choice for applications requiring flexibility in task distribution.

Fork-Join and Executor-based methods can be optimal in different scenarios depending on picture size and workload characteristics. Here's an overview of where each method shines:

Fork-Join

Fork-Join is best suited for smaller photo dimensions where tasks can be dynamically partitioned and executed in parallel. It efficiently manages task distribution and completion, making it ideal for smaller workloads. The specific cases where Fork-Join is the optimal choice include:

Smaller Photo Dimensions:

For dimensions like 696x522 and 1920x1080, Fork-Join provides the best performance. This is because the workload can be divided into smaller chunks and processed concurrently, with tasks completing quickly and resources being reused efficiently.

Dynamic Task Distribution:

Fork-Join's dynamic task partitioning is effective for workloads with varying execution times or tasks that can be completed quickly. This adaptability ensures optimal resource utilization and reduced overhead.

Executor-Based

Executor-based methods are generally best for larger photo dimensions, particularly when tasks can be partitioned for parallel execution. This approach provides efficient thread management, making it suitable for larger workloads. The specific cases where Executor-based methods, especially with partitions, are optimal include:

Larger Photo Dimensions:

For dimensions like 5472x3648 and 7680x4320, Executor-Based with Partitions delivers the best results. This method effectively divides the workload into smaller partitions, allowing for concurrent processing and optimal utilization of system resources.

Complex Workloads:

Executor-based methods with partitions excel when the workload is more complex or when tasks require synchronization or coordination. This structure provides flexibility while managing overhead effectively.

In summary, Fork-Join is ideal for smaller photo dimensions and scenarios where dynamic partitioning is beneficial, while Executor-Based with Partitions is best for larger photo dimensions and complex workloads where parallel execution and efficient thread management are key to performance.

Although Completable Future with Partitions is the third-best in terms of total time, it provides benefits through its asynchronous nature and effective task management. This approach can be useful in scenarios where asynchronous operations are required or when tasks have complex dependencies.

Regarding garbage collectors, **G1 GC** is the most consistent and efficient, delivering balanced performance across different execution methods and photo dimensions. This GC offers an optimal balance between garbage collection pause times and throughput, making it suitable for high-performance applications.

Overall, the best approach combines **Executor-Based with Partitions with G1 GC** for optimal performance, providing a balance of speed, resource management, and adaptability to varying workloads. This combination is ideal for applications where high throughput and efficient resource utilization are critical.

4.2 Rafael's PC

Rafael computer's specs are: 8 logic processors, processor ARM apple M1 with 8 GB of ram.

4.2.1 Sequential

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	23	19	14	24	98	7	185
1920 x 1080	17	14	43	58	472	60	664
5472 x 3648	1698	969	373	546	7029	129	10744
7680 x 4320	2605	3098	576	851	11442	884	19456

Table 30 - Parallel GC Sequential

Shenandoah GC:

	Shenandoah GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	11	9	17	26	96	8	167
1920 x 1080	22	18	45	61	520	66	732
5472 x 3648	184	259	392	572	7747	148	9302
7680 x 4320	419	415	569	1000	15895	1386	19684

Table 31 - Shenandoah GC Sequential

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	16	9	15	24	110	8	182
1920 x 1080	36	32	45	59	540	87	799
5472 x 3648	551	324	359	531	7069	345	9179
7680 x 4320	827	868	604	1523	10307	1143	15272

Table 32 - G1 GC Sequential

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	11	9	15	25	99	8	167
1920 x 1080	24	19	45	63	525	76	752
5472 x 3648	426	367	390	907	10035	917	13042
7680 x 4320	679	2824	948	2516	20438	2238	29643

Table 33 - Z GC Sequential

4.2.2 Multithread

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	33	21	28	17	39	8	146
1920 x 1080	66	54	47	43	146	48	404
5472 x 3648	1378	1305	95	185	2123	77	5163
7680 x 4320	2357	2069	155	279	4311	401	9572

Table 34 - Parallel GC Multithread

Shenandoah GC:

	Shenandoah GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	31	22	29	32	32	12	158
1920 x 1080	10	9	39	16	107	27	208
5472 x 3648	247	248	102	191	4574	104	5466
7680 x 4320	477	350	139	476	12968	752	15162

Table 35 - Shenandoah GC Multithread

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	24	21	25	20	39	9	138
1920 x 1080	49	67	47	43	139	53	398
5472 x 3648	650	602	124	298	2430	218	4322
7680 x 4320	1021	877	155	404	4405	541	7403

Table 36 - G1 GC Multithread

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	31	26	27	30	32	13	159
1920 x 1080	63	55	65	47	140	39	409
5472 x 3648	542	638	114	430	5979	463	8166
7680 x 4320	1151	1932	273	617	17695	961	22629

Table 37 - Z GC Multithread

4.2.3 Threadpool Executor Based

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	5	29	26	19	29	4	112
1920 x 1080	6	7	33	13	135	17	211
5472 x 3648	540	160	86	173	2444	260	3663
7680 x 4320	411	930	122	318	2888	572	5241

Table 38 - Parallel GC Threadpool Executor Based

Shenandoah GC:

	Shenandoah GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	29	32	25	26	29	3	144
1920 x 1080	10	9	39	16	107	27	208
5472 x 3648	184	200	93	180	4106	108	4871
7680 x 4320	467	266	250	376	5023	640	7022

Table 39 - Shenandoah GC Threadpool Executor Based

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	8	6	33	18	22	5	92
1920 x 1080	9	10	47	14	135	43	258
5472 x 3648	644	367	92	271	2236	172	3782
7680 x 4320	1027	1000	258	346	3901	523	7055

Table 40 - G1 GC Threadpool Executor Based

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	32	4	31	30	37	3	137
1920 x 1080	10	13	32	18	120	22	215
5472 x 3648	569	551	95	232	5028	432	6907
7680 x 4320	765	1872	179	620	5012	1074	9522

Table 41 - Z GC Threadpool Executor Based

4.2.4 Threadpool Executor Based with 16 Partitions

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	1	2	2	2	20	1	28
1920 x 1080	5	8	9	16	114	14	166
5472 x 3648	530	495	62	133	2897	122	4239
7680 x 4320	615	1071	109	257	5401	923	8376

Table 42 - Parallel GC Threadpool Executor Based with 16 Partitions

Shenandoah GC:

	Shenandoah GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	2	2	2	2	23	2	33
1920 x 1080	5	5	7	11	98	13	139
5472 x 3648	143	238	65	158	4517	45	5166
7680 x 4320	416	389	254	339	5016	424	6838

Table 43 - Shenandoah GC Threadpool Executor Based with 16 Partitions

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	1	1	2	2	22	2	30
1920 x 1080	5	6	7	10	107	19	154
5472 x 3648	217	365	68	120	1616	119	2505
7680 x 4320	822	697	152	308	3389	356	5724

Table 44 - G1 GC Threadpool Executor Based with 16 Partitions

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	2	1	3	2	22	3	33
1920 x 1080	9	38	8	12	184	40	291
5472 x 3648	525	536	93	318	5015	486	6973
7680 x 4320	778	2145	249	545	5009	1762	10488

Table 45 - Z GC Threadpool Executor Based with 16 Partitions

4.2.5 Threadpool Fork Join Based (100 threshold)

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	2	2	7	3	21	2	37
1920 x 1080	6	18	13	11	118	15	181
5472 x 3648	366	440	78	128	1728	163	2903
7680 x 4320	462	1006	158	327	3605	523	6081

Table 46 - Parallel GC Threadpool Fork Join Based (100 threshold)

Shenandoah GC:

	Shenandoah GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	2	1	11	2	25	2	43
1920 x 1080	6	6	20	11	111	13	167
5472 x 3648	138	259	87	122	4501	44	5151
7680 x 4320	405	516	250	355	15011	248	16785

Table 47 - Shenandoah GC Threadpool Fork Join Based (100 threshold)

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	3	1	17	3	26	3	53
1920 x 1080	15	6	21	11	127	17	197
5472 x 3648	110	237	91	124	1968	102	2632
7680 x 4320	664	650	132	419	3908	481	6254

Table 48 - G1 GC Threadpool Fork Join Based (100 threshold)

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	4	3	9	2	31	3	52
1920 x 1080	24	25	16	12	227	40	344
5472 x 3648	564	556	85	222	6774	323	8524
7680 x 4320	1133	1862	236	527	18684	987	23429

Table 49 - Z GC Fork Join Based (100 threshold)

4.2.6 Threadpool Comparable Future Based 16 Partitions

Parallel GC:

	Parallel GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	5	2	2	3	23	3	38
1920 x 1080	11	6	9	13	107	17	163
5472 x 3648	118	270	91	132	2522	143	3276
7680 x 4320	790	454	110	340	4097	281	6072

Table 50 - Parallel GC Threadpool Comparable Future Based 16 Partitions

Shenandoah GC:

	Shenandoah GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	3	3	3	3	23	4	39
1920 x 1080	7	6	9	12	103	16	153
5472 x 3648	146	189	68	131	4403	44	4981
7680 x 4320	446	549	191	384	12998	294	14862

Table 51 - Shenandoah GC Threadpool Comparable Future Based 16 Partitions

G1 GC:

	G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	3	3	2	4	22	3	37
1920 x 1080	9	6	9	12	120	17	173
5472 x 3648	329	490	85	160	1812	182	3058
7680 x 4320	771	697	115	518	4479	406	6986

Table 52 - G1 GC Threadpool Comparable Future Based 16 Partitions

Z GC:

	Z GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	8	3	2	3	27	3	46
1920 x 1080	31	26	9	13	141	18	238
5472 x 3648	650	543	87	166	6710	431	8587
7680 x 4320	890	1897	174	800	17108	744	21613

Table 53 - Z GC Threadpool Comparable Future Based 16 Partitions

4.2.7 General Results

Total Time per GC

GC	Parallel	Shenandoah	G1	Z
Total Time	87121	117481	76683	172366

Table 54 - Total Time per GC

Total Time per Executor Method

Executor	Sequential	Multithread	Executor	Executor Partition	ForkJoin	CompletableFuture
Nº of best results	0	0	5	13	1	1
Total Time	129970	79903	49440	51183	72833	70322

Table 55 - Total Time per Executor Method

Total Time Executor Method per Image Size

	Sequential	Multithread	Executor	Executor Partition	ForkJoin	CompletableFuture
696 x 522	701	601	485	124	185	160
1920 x 1080	2947	1419	892	750	889	727
5472 x 3648	42267	23117	19223	18883	19210	19902
7680 x 4320	84055	54766	28840	31426	52549	49533

Table 56 - Total Time of each method per Image Size

4.2.8 Partitions and Thresholds Variation

4.2.8.1 Partition Variation with the better GC

8 Partitions

	8 partitions – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	1	1	2	3	27	3	37
1920 x 1080	5	5	8	11	106	27	162
5472 x 3648	379	359	69	215	2087	152	3261
7680 x 4320	752	933	139	303	4179	415	6721

Table 57 - 8 Partitions with G1 GC

16 Partitions

	16 partitions – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	1	1	2	2	22	2	30
1920 x 1080	5	6	7	10	107	19	154
5472 x 3648	217	365	68	120	1616	119	2505
7680 x 4320	822	697	152	308	3389	356	5724

Table 58 - 16 Partitions with G1 GC

32 Partitions

	32 partitions – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	2	1	1	2	23	2	31
1920 x 1080	5	5	8	10	110	15	153
5472 x 3648	106	273	65	113	1583	33	2173
7680 x 4320	943	771	137	229	3748	336	6164

Table 59 - 32 Partitions with G1 GC

Results

Partitions	8	16	32
Total	10181	8413	8521
	3º	1º	2º

Table 60 - Final results of partition tests

4.2.8.2 Threshold Variation with the better GC

Threshold 50

	Thershold 50 – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	2	2	13	2	22	2	43
1920 x 1080	15	11	32	11	112	14	195
5472 x 3648	425	544	90	164	1710	154	3087
7680 x 4320	702	718	151	237	4103	389	6300

Table 61 - 50 Threshold with G1 GC

Threshold 100

	Thershold 100 – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	3	1	17	3	26	3	53
1920 x 1080	15	6	21	11	127	17	197
5472 x 3648	110	237	91	124	1968	102	2632
7680 x 4320	664	650	132	419	3908	481	6254

Table 62 - 100 Threshold with G1 GC

Threshold 150

	Thershold 150 – G1 GC						
	Brighter	Grayscale	Glass	Swirl	Blur	Conditional Blur	Total
696 x 522	2	2	13	2	25	3	47
1920 x 1080	7	6	16	12	130	17	188
5472 x 3648	175	474	86	171	1510	31	2447
7680 x 4320	385	585	165	240	4228	358	5961

Table 63 - 150 Threshold with G1 GC

Results

Threshold	150	100	50
Total	8643	9136	9625
	1º	2º	3º

Table 64 - Final results of the Threshold tests

4.2.9 Conclusions

4.2.9.1 Garbage Collector Performance

Total Times Across Different Garbage Collectors (GC)

These tests were done with a max heap size of 6 GB of memory.

Parallel GC: Performs well with larger photo dimensions, but its total time is higher compared to G1. This suggests it might have more overhead or less efficient memory management. In some situations, it even has better performance than the G1 on large photos.

Shenandoah GC: Performs well on small to medium size photos, but starts falling down the order if the photo dimensions start getting bigger. This issue happens because its performance is dependent on the heap size and since my computer only has 8gb of Ram, with a max of 6 GB of Ram of heap size it can reduce the performance. Another possible issue is that since Shenandoah is a concurrent GC, it employs barriers to maintain invariants during the collection cycle. Those barriers might induce the measurable throughput loss.

G1 GC: Generally consistent across different dimensions and execution methods. Its balanced performance makes it a strong choice for high-throughput applications.

Z GC: Shows the longest total time, indicating it may not be ideal for workloads requiring high performance. Z GC focuses on ultra-low latency but might not be optimized for high throughput.

4.2.9.2 Execution Methods Comparison

4.2.9.2.1 Sequential Execution

Sequential processing is consistently slower compared to other methods, indicating that threading or parallel execution provides a significant speedup. It's not suitable for applications requiring high performance.

4.2.9.2.2 Multithreaded Execution

Multithreaded execution is faster than sequential, but not the fastest overall. It improves performance, but there's potential for optimization through better task distribution and management.

4.2.9.2.3 Executor-Based Threadpool

Without Partitions: Generally, performs well, especially on larger photo dimensions. It leverages multiple threads but might not optimize task distribution as effectively as partitioned approaches.

With Partitions: Shows better performance compared to the unpartitioned approach, on most photos. Partitioning allows for more efficient task distribution, reducing overhead and improving performance.

Difference Between with and Without Partitions

Partitioning allows tasks to be divided into smaller, manageable units, enabling more efficient use of multiple threads. This leads to reduced overhead and improved performance compared to non-partitioned methods.

4.2.9.2.4 Fork-Join

Performs well, especially on smaller photo dimensions. The dynamic partitioning nature of Fork-Join allows it to break down tasks and allocate them as needed, optimizing parallel execution.

4.2.9.2.5 Completable Future with Partitions

This method tends to perform great across all photo dimensions, demonstrating the efficiency of using partitioning with completable futures for asynchronous task management. This approach can dynamically distribute tasks and manage dependencies effectively. It has similar numbers to fork-join on most scenarios, which makes sense since both Fork Join Task and Completable Future use Fork Join Pool.

4.2.10 Threshold and Partition Variations

4.2.10.1 Threshold Variation

Testing with different thresholds (50, 100, 150) using G1 GC revealed that Threshold 150 is the most consistent across all tests. It provides a balanced compromise between processing time and resource management, yielding optimal performance.

4.2.10.2 Partition Variations

16 Partitions: Achieves the lowest total time (8413 ms), indicating it's the most efficient configuration among the tested partition counts. It strikes a good balance between parallelism and thread management overhead.

8 Partitions: Has the highest total time (10181 ms). While fewer partitions can reduce overhead, it might limit the advantages of parallelism, leading to increased total times.

32 Partitions: Has a lower total time than 8 partitions but is still higher than 16 partitions (8521 ms). This could be due to increased overhead from managing more partitions, offsetting gains from additional parallelism.

Thus, in the context of running filters on a photo, 16 partitions seem to offer the best balance between parallelism and management overhead. This suggests that partitioning plays a significant role in performance, with a moderate number of partitions yielding better results.

4.2.11 Analysis and Expectations of The Best Results

Partitioning and Executor-Based Methods

Partitioning allows tasks to be spread across multiple threads efficiently, reducing idle time and allowing better use of CPU cores. Executor-based methods manage thread allocation, contributing to better performance when combined with partitioning.

Fork-Join

Fork-Join's dynamic partitioning capability helps optimize performance, especially for tasks with variable execution times. This method can adapt to the workload's demands, enhancing efficiency.

Garbage Collectors

G1 GC's consistent performance can be attributed to its efficient memory management across larger workloads, whereas Parallel GC may have more overhead due to its design. This could lead to reduced performance when handling large workloads.

4.2.12 Conclusion

Based on the analysis, the most efficient method for running filters on photos is **Executor-Based with Partitions**. This method outperformed other execution approaches, offering the best total time across various photo dimensions. Its efficiency stems from partitioning tasks into smaller units and executing them concurrently, which maximizes the use of system resources while minimizing overhead.

Fork-Join and Executor-based methods can be optimal in different scenarios depending on picture size and workload characteristics. Here's an overview of where each method shines:

Fork-Join

Fork-Join is best suited for smaller photo dimensions where tasks can be dynamically partitioned and executed in parallel. It efficiently manages task distribution and completion, making it ideal for smaller workloads. The specific cases where Fork-Join is the optimal choice include:

Smaller Photo Dimensions:

For dimensions like 696x522 and 1920x1080, Fork-Join provides pretty good performance. This is because the workload can be divided into smaller chunks and processed concurrently, with tasks completing quickly and resources being reused efficiently.

Dynamic Task Distribution:

Fork-Join's dynamic task partitioning is effective for workloads with varying execution times or tasks that can be completed quickly. This adaptability ensures optimal resource utilization and reduced overhead.

Executor-Based

Executor-based methods are generally best for larger photo dimensions, particularly when tasks can be partitioned for parallel execution. This approach provides efficient thread management, making it suitable for larger workloads. The specific cases where Executor-based methods, especially with partitions, are optimal include:

Larger Photo Dimensions:

For dimensions like 5472x3648 and 7680x4320, Executor-Based with Partitions delivers the best results. This method effectively divides the workload into smaller partitions, allowing for concurrent processing and optimal utilization of system resources.

Complex Workloads:

Executor-based methods with partitions excel when the workload is more complex or when tasks require synchronization or coordination. This structure provides flexibility while managing overhead effectively. In most scenarios, this method seems to deliver the best results.

In summary, Fork-Join is ideal for smaller photo dimensions and scenarios where dynamic partitioning is beneficial, while Executor-Based with Partitions is good for all types of photos but the best for larger photo dimensions and complex workloads where parallel execution and efficient thread management are key to performance.

Although Completable Future with Partitions is the third-best in terms of total time, it provides benefits through its asynchronous nature and effective task management. This approach can be useful in scenarios where asynchronous operations are required or when tasks have complex dependencies.

Regarding garbage collectors, **G1 GC** is the most consistent and efficient, delivering balanced performance across different execution methods and photo dimensions. This GC offers an optimal balance between garbage collection pause times and throughput, making it suitable for high-performance applications.

Overall, the best approach combines **Executor-Based with Partitions with G1 GC** for optimal performance, providing a balance of speed, resource management, and adaptability to varying workloads. This combination is ideal for applications where high throughput and efficient resource utilization are critical.

5 Conclusion

5.1 Garbage Collector Performance

Oscar's PC

- Parallel GC: Performs well with larger photo dimensions but has higher total times compared to G1. This may be due to its design, leading to more overhead or less efficient memory management.
- G1 GC: Offers consistent performance across different image sizes and execution methods. It’s a balanced choice for high-throughput applications.
- Z GC: Has the longest total times, indicating it's not ideal for high-performance workloads. It focuses on ultra-low latency but may not be optimized for high throughput.

Rafael's PC

- Parallel GC: Works best with larger photo dimensions, sometimes outperforming G1 GC. This could be due to variations in overhead or garbage collection efficiency.
- Shenandoah GC: Performs well on smaller photos but struggles with larger ones. This may be due to its dependency on heap size, affecting its ability to manage large workloads effectively.
- G1 GC: Like Oscar's PC, G1 GC provides consistent performance across different image sizes and execution methods. It's suitable for high-throughput applications.
- Z GC: Shows the longest total times among the garbage collectors, especially with larger photo dimensions. This could indicate less efficiency in high-throughput environments.

5.2 Execution Methods Comparison

Oscar's PC

- Sequential: Consistently slower, indicating the need for threading or parallel execution to improve performance.
- Multithreaded: Faster than sequential, but not the most efficient. There's potential for optimization through better task distribution.
- Executor-Based with Partitions: Generally, the best approach for running filters on photos, particularly with larger photo dimensions. It maximizes resource utilization by partitioning tasks for concurrent execution.
- Fork-Join: Ideal for smaller photo dimensions due to its dynamic partitioning. This method adapts to the workload's needs, optimizing parallel execution.

Rafael's PC

- Sequential: Like Oscar's PC, sequential processing is consistently slower, reinforcing the need for parallel execution.
- Multithreaded: Provides a speed improvement over sequential but can be further optimized.
- Executor-Based with Partitions: Delivers the best results, especially for larger photo dimensions. The partitioning helps spread tasks across multiple threads for optimal performance.
- Fork-Join: Best suited for smaller photo dimensions, demonstrating good performance due to its dynamic task distribution.

5.3 Threshold and Partition Variations

Oscar's PC

- 16 Partitions: Achieves the best total time, indicating it's the most efficient configuration. This setup strikes a good balance between parallelism and thread management overhead.
- Threshold 100: Provides the most consistent results, suggesting that this level offers a balanced approach to processing time and resource management.

Rafael's PC

- 16 Partitions: Again, this configuration offers the best performance, like Oscar's PC. It balances parallelism and management overhead effectively.
- Threshold 150: Yields the most consistent results, suggesting that it’s optimal for managing resource utilization and processing time.

5.4 Overall Analysis and Expectations

The performance results suggest that **Executor-Based with Partitions** is the best approach for running filters on photos, especially for larger photo dimensions. This method effectively distributes tasks, reducing overhead and maximizing resource utilization. It also adapts well to complex workloads where parallel execution and efficient thread management are crucial.

Fork-Join is optimal for smaller photo dimensions and dynamic workloads, providing flexibility through its dynamic task distribution. **Sequential** execution is consistently slower, highlighting the importance of parallel processing.

Regarding garbage collectors, G1 GC is the most consistent and efficient across both PCs, offering a good balance of throughput and low pause times. Parallel GC performs well in some cases but may have higher overhead, while Z GC and Shenandoah GC struggle with high-throughput workloads, especially with larger photo dimensions.

5.5 Final Thoughts

To achieve optimal performance, use Executor-Based with Partitions for larger photo dimensions, and Fork-Join for smaller dimensions. Pair these methods with G1 GC for garbage collection, which provides a balanced and efficient approach to managing memory. This combination is ideal for applications where high throughput and efficient resource utilization are required.

6 Appendix

6.1 Appendix – Glossary

- FYI - For Your Information.
- G1 -A Java garbage collector designed for low-latency applications.
- GB -Gigabyte.
- GC - Garbage Collector.
- HD - High Definition.
- JAR - Java Archive
- PC - Personal Computer
- SISMD - Sistemas Multinúcleo e Distribuídos.
- Z -A Java garbage collector.
- ZIP - A compressed file format used for archiving and reducing file sizes.