A Performance Analysis of Modern Garbage Collectors in the JDK 20 Environment

By

Md. Jahidul Islam
Student Id: 0421052108

A.T.M Mizanur Rahman
Student Id: 0422052038

Submitted to
Dr. Rifat Shahriyar
Professor

in partial fulfillment of the requirements
for the course works of CSE6305: Programming Languages and Systems

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET), Dhaka 1000, Bangladesh

October 2022

A Performance Analysis of Modern Garbage Collectors in the JDK 20 Environment

MD. JAHIDUL ISLAM* and A.T.M MIZANUR RAHMAN†, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Bangladesh

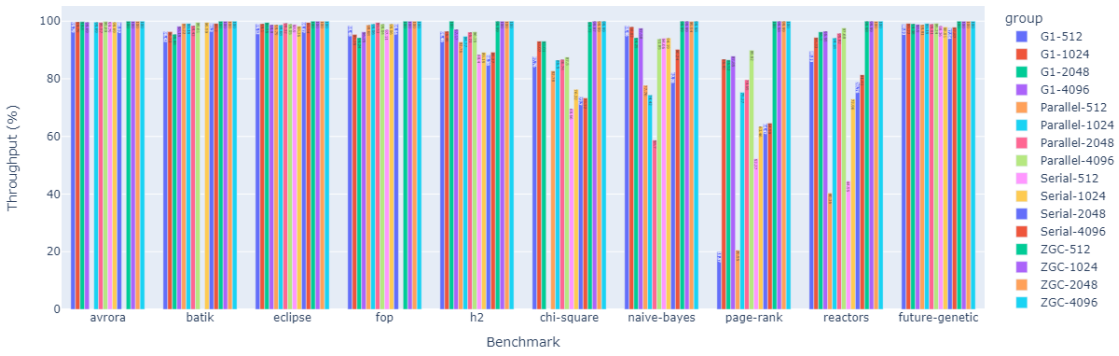


Fig. 1. Throughput percentage of various GC algorithms

Garbage collection is a fundamental aspect of Java Virtual Machine (JVM) memory management, and choosing the optimal garbage collector is essential for attaining optimal application performance. In this work, we conduct experiments with the benchmarks from DaCapo and Renaissance benchmark suites in both fixed and variable heap environments to determine the efficacy of JDK 20's garbage collectors. The ZGC algorithm has the highest throughput and the shortest pause periods, whereas the Serial GC algorithm has the lowest throughput and the longest pause intervals. Additionally, it is discovered that the G1 algorithm manages the previous generation heap and metaspace less efficiently. Our research offers valuable insights into the efficacy of garbage collection algorithms and can aid application developers in selecting the optimal garbage collection algorithm. Our investigation can be expanded by analyzing additional benchmark suites and garbage collection algorithms across a broader range of heap sizes in future work.

Additional Key Words and Phrases: Performance Analysis, Garbage Collectors, Serial GC, Parallel GC, ZGC, G1 GC

*Student Id: 0421052108
†Student Id: 0422052038

Authors' address: Md. Jahidul Islam, 0421052108@grad.cse.buet.ac.bd; A.T.M Mizanur Rahman, 0422052038@grad.cse.buet.ac.bd, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, 1000, Bangladesh.

Manuscript submitted to ACM

ACM Reference Format:
Md. Jahidul Islam and A.T.M Mizanur Rahman. 2023. A Performance Analysis of Modern Garbage Collectors in the JDK 20 Environment. 1, 1 (April 2023), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Memory management is an essential concept in computer programming that deals with the allocation and deallocation of memory resources for the execution of programs. Memory management also refers to the process of managing memory. It entails the act of reserving memory space in order to store data, as well as recovering that data when it is required, and then deallocating that memory space when it is no longer needed. Memory management is an essential aspect to think about when working with high-level and low-level programming languages alike, and it has the potential to have a considerable bearing on both the speed and the reliability of software systems.

Memory management is taken care of by the Java Virtual Machine (JVM), which is used by high-level programming languages such as Java. The JVM is an essential component of the Java Runtime Environment (JRE) that is responsible for executing Java bytecode. Additionally, the JVM is responsible for managing memory allocation and deallocation through a process known as garbage collection. Programmers can concentrate on developing code without having to be concerned about manually managing memory according to a process known as garbage collection, which is an automatic procedure that reclaims memory space that has been utilized.

The garbage collector of the JVM performs periodic scans of the heap memory in order to locate objects in the memory that are no longer being used by the program. After that, it frees up the memory space that these things were occupying and makes that memory space accessible for later usage. The garbage collector's algorithms are designed to minimize the impact on the program's execution time and prevent memory leaks.

Memory allocated for heaps is an integral component of the Java Virtual Machine's memory management. The objects that are produced by a Java application are saved in a section of memory referred to as heap memory. The JVM does a dynamic allocation of it, and the size of it can be changed according to the requirements of the program. Eden space, Survivor space, and Tenured space are some of the regions that make up heap memory, each of which has its own unique allocation method. Heap memory is split up into several separate regions. It is the job of the garbage collector to release heap memory that is no longer being utilized by a program when it is no longer needed.

Memory management is something that is managed manually by the programmer in low-level programming languages such as C and C++. This indicates that it is the responsibility of the programmer to allocate and deallocate memory prior to the execution of the program. Memory can be allotted in these languages through the use of functions like `malloc()` and `calloc()`, and it can be deallocated through the use of the `free()` function. Memory leaks are caused when software fails to release memory that is no longer being used and might be the result of improper memory management. Memory leaks can cause programs to crash. Memory leaks can result in poor performance and instability for a program, as they can cause the application to crash or slow down over time.

Dangling pointers are yet another typical problem that can develop as a consequence of inefficient memory management in low-level programming languages. A dangling pointer is a pointer that points to a memory region that has been deallocated. This causes the program to behave in an unpredictable manner whenever it attempts to access the memory address in question. It can be challenging to find dangling pointers and even more challenging to correct them, but both of these problems can result in programs crashing or producing inaccurate results.

For effective, secure, and reliable software development, memory management, and garbage collection must be studied and analyzed. The efficient use of resources, increased speed and responsiveness of a program, and stability

can all be achieved with good memory management. The influence on a program's execution time can be minimized, and memory use can be decreased, with effective garbage collection. Cyber attacks frequently target memory-related weaknesses, therefore learning about memory management and garbage collection can aid in the development of secure coding techniques. For the purpose of creating portable software programs, it is additionally required to research and analyze the various memory management and garbage collection processes that are present across various programming languages and platforms.

With an emphasis on contemporary garbage collectors, this project intends to investigate memory management in Java Development Kit version 20 (JDK 20). The study will examine several modern garbage collection methods, their benefits and drawbacks, and how they affect the efficiency and stability of programs. The objective is to enhance Java application memory management strategies for more effective and reliable software development.

This project report is organized into the following sections. The background information on memory management, garbage collection, and their importance to software development are covered in Section 2, along with our justification for doing this study. The methodology for this study, including our experimental setup and the particular garbage collectors and benchmarks employed, is covered in full in Section 3. The performance analysis of this project's work is covered in Section 4, which also outlines the findings of our studies and their implications for Java applications' memory management. In Section 5, which comes to a close, we offer summaries of the study's findings and considerations of how they may affect future memory management and garbage collection studies.

2 BACKGROUNDS AND MOTIVATIONS

Understanding the various components of the Java Virtual Machine is essential for efficient memory management and optimal performance of Java applications. The Java Virtual Machine (JVM) is a complex system, as shown in "Fig. 2" composed of multiple components that manage memory and execute code in concert. Class Loader Subsystem, which loads and initializes classes during runtime, is one of the fundamental components of memory management. This subsystem consists of three distinct categories of class loaders, each with a distinct set of responsibilities. The Runtime Data Area, which includes regions such as the method area, heap, java threads, program counter registers, and native internal threads, is an essential component of the Java Virtual Machine. The method area contains class definitions and static variables, whereas the heap manages and allocates objects. The execution engine, which consists of the Just-in-Time (JIT) compiler and the garbage collector, is another crucial component that executes code. The JIT compiler converts bytecode to machine code, which improves application performance, whereas the garbage collector manages memory by releasing space when objects are no longer required. In addition, the native method interface enables Java code to interact with native libraries written in other programming languages, granting access to system resources and low-level operations that are unavailable in Java.

2.1 Heap Memory

Heap memory, also referred to as dynamic memory allocation, is the component of a computer's memory where programs can request memory space to be allocated dynamically during runtime. In contrast to stack memory, heap memory is allocated and deallocated manually by the programmer. The heap is a sizable memory block that is not managed automatically by the operating system and is available to any program that requests it. Heap memory is beneficial when a program must allocate a large amount of memory, such as when working with complex data structures or when the required memory size is unknown at compile time. However, improper management of heap memory

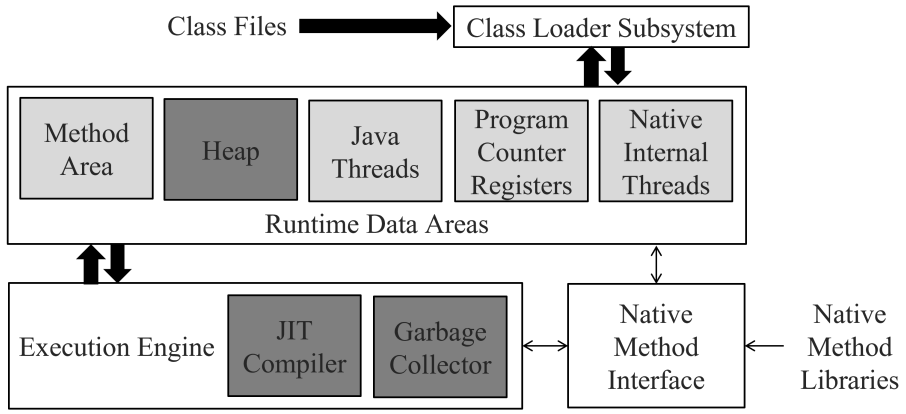


Fig. 2. Components of Java Virtual Machine related to memory management.[6]

can result in memory leaks and other issues, so programmers must be conversant with heap memory management concepts.

There are two varieties of heap space in Java Virtual Machines (JVMs): traditional heap space and G1 heap space as shown in "Fig 3". The conventional heap space comprises of a contiguous memory block divided into a young generation and an old generation. Young generation has access to both Eden space and survivor space. As objects are referenced, they are created in Eden space and transferred to the survivor space. When survivor space is at capacity, objects are transferred to the previous generation. However, traditional heap space has a number of drawbacks, including lengthy garbage collection times and memory fragmentation, which contribute to an increase in memory consumption. In response to these issues, the G1 heap space was developed. G1 heap space partitions memory into regions capable of independent garbage collection. When G1 heap space executes garbage collection, it collects only the regions containing the most garbage, thereby minimizing its impact on application performance. Additionally, G1 heap space can defragment memory, resulting in more efficient memory utilization.

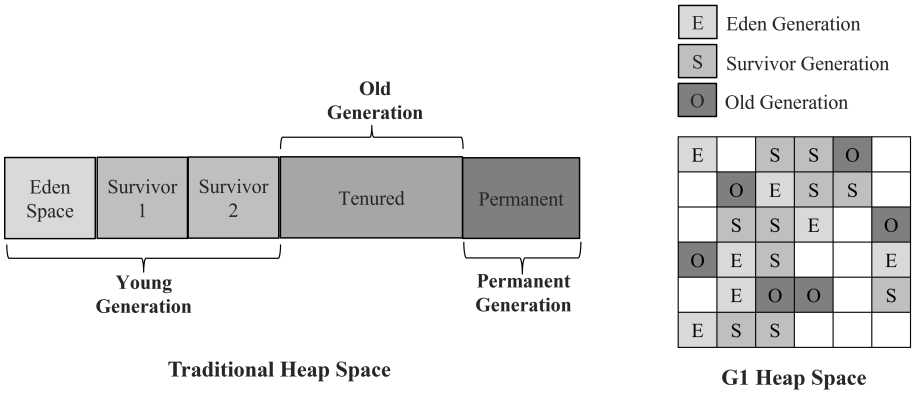


Fig. 3. Heap space in Java Virtual Machines.[5]

2.2 Garbage Collection

Garbage collection is an automatic memory management procedure within a program. It identifies and removes superfluous objects from the heap in order to make room for additional objects. Mark and Sweep [10] is one of the earliest and most fundamental techniques used by garbage collectors. The two phases of this algorithm are labeling and sweeping. During the marking phase, the garbage collector examines all living objects in the heap and labels those that are alive. During the sweeping phase, the garbage collector releases memory occupied by unmarked objects. Garbage collection has many benefits, including automatic memory management that saves time and effort for developers.

Garbage collection algorithms, also known as garbage collectors, employ a variety of techniques to detect and collect garbage. Other garbage collectors prioritize throughput and efficiency over minimizing latency. Garbage collectors do not, however, solve all issues associated with autonomous memory management. They perform multiple duties, including allocating memory from and releasing it to the operating system, assigning memory to the application as requested, identifying the portions still in use by the application, and reclaiming unused memory for use by the application in the future. [7]

Several garbage collection algorithms are used to enhance the garbage collection process in Java Virtual Machines. The Concurrent Mark and Sweep algorithm is intended to reduce pause times, allowing applications to continue operating during garbage collection. Using multiple threads to perform garbage collection, the Parallel Collector algorithm is intended to increase throughput. G1 collector [4] is a more recent algorithm that divides the heap into regions for more effective garbage collection. Overall, garbage collection algorithms play a crucial role in managing memory and assuring the efficient operation of a program.

2.3 JDK 20 GCs

In JDK 20, several garbage collectors are available, each designed to provide varying performance, memory consumption, and pause time benefits. Here are the available garbage collectors in JDK 20:

Serial GC This garbage collector is a simple, single-threaded garbage collector designed for small applications or those with minimal memory requirements. It is appropriate for desktop and client applications that require minimal latency and do not have a high memory footprint.

Parallel GC The Parallel Garbage Collector, also known as the Throughput Collector, is intended for server applications that demand a high throughput and can tolerate longer pause periods. It is multithreaded, enabling it to perform garbage collection in parallel, and it can manage heaps that are larger.

CMS GC It is also known as the Concurrent Mark Sweep Collector, is designed for applications that require low-latency garbage collection. It utilizes multiple threads to collect garbage in parallel, minimizing pauses. Ideal for large applications requiring minimal latency and high throughput.

G1 GC It is also known as the Garbage-First Collector, is designed for large collections and applications with high throughput. Using region-based memory management and concurrent garbage collection, it is able to minimize delay times. It is also designed to respond to the application's behavior by modifying its collection strategy.

ZGC The Z Garbage Collector, also known as ZGC, is a scalable garbage collector designed for applications requiring minimal pause times and large heaps. It collects garbage concurrently and in parallel, minimizing pause periods even for extremely large heaps. It is designed to function effectively in a wide range of environments, from modest devices to large cloud environments.

Shenandoah GC It is a concurrent garbage collector designed for large heaps and short pause periods. It collects garbage concurrently and in parallel, minimizing pause periods even for extremely large heaps. It is designed to function effectively in a variety of environments and is especially adapted for cloud-based applications.

Epsilon GC Unlike the other garbage collectors, is not production-ready and should only be used for testing and diagnostic purposes. It is intended to be a garbage collector that performs no memory reclamation. This is beneficial for performance analysis or testing scenarios in which memory consumption is not a concern.

2.4 Motivations and Contributions

Numerous performance analyses on garbage collection using older versions of the Java Development Kit (JDK) and numerous garbage collectors have been conducted in previous studies. These studies also investigated the impact of memory management on the efficacy of the Java Virtual Machine (JVM). They investigated the effectiveness of various memory management techniques in reducing memory consumption and enhancing application performance. The studies have contributed to the comprehension of how memory management and garbage collection optimize the use of system resources by Java applications.

In [9] authors examine the essential components of the Java Virtual Machine (JVM) in the context of automated memory management by means of Garbage Collection (GC) algorithms. This article discusses the various garbage collection (GC) algorithms supported by the JVM, the ongoing development of the Java Development Kit (JDK) and new Java Enhancement Proposals (JEPs), and the need to optimize GC algorithms to enhance application performance. Using benchmarking applications from the DaCapo suite, the author evaluates the efficacy of numerous GC algorithms. They provide a comprehensive summary of the JVM's role in memory management and its ongoing evolution to satisfy the changing needs of developers.

Authors investigate the function of garbage collection in managed runtime environments, such as the Java Virtual Machine (JVM) on large-scale multicore servers in [3]. The authors studied all available Java garbage collectors and discovered that their behavior varies depending on the environment. ParallelOld, the default Java garbage collector, is stable and adequate in benchmark environments but causes unacceptable delays in real-world scenarios. In a client-server environment, G1 and ConcurrentMarkSweep garbage collectors perform better than ParallelOld, but they can still have a significant impact on response time. The research emphasizes the significance of selecting the optimal refuse collector for a given environment in order to optimize performance and minimize downtime.

In [1], the authors investigate the effect of selecting the appropriate automated memory management strategy on application performance, focusing on Java applications running on the Java Virtual Machine. (JVM). In 2017, the Garbage First (G1) garbage collector has become the industry standard. However, other garbage collectors, such as Shenandoah GC and Z Garbage Collector (ZGC), have evolved and were recently promoted from experimental to production-ready status. The research emphasizes the need for a more comprehensive examination of the performance differences between these standard GCs using experimental benchmarks and real-world use cases. The focus of this paper is a comparative analysis of the efficacy of the default G1 and the Shenandoah and Z garbage collectors in managing various memory issues, heap allocation, CPU utilization, and time consumption.

However, there is still an opportunity to study the various garbage collection algorithms in the most recent Java Development Kit (JDK 20). Our contributions to this study include:

- We provide specifics on the study related to the performance evaluation of garbage collectors that are available in JDK 20.

- We analyse the efficacy of garbage collection based on various benchmarks, including big-data benchmarks, in a high-configuration environment.

Considering time, complexity, and environmental setup, we only evaluate GC algorithms on 12 DaCapo benchmarks for fixed heap-based experiments and 5 benchmarks for variable heap-based experiments. Similarly, for the Renaissance benchmark suite, we use 9 benchmarks for fixed heap-based experiments and 5 benchmarks for heap-based experiments with varying heap sizes. We consider regular benchmark applications and big-data benchmark applications for this analysis. We use heap capacities of 512 MB, 1024 MB, 2048 MB, and 4096 MB for various heap environments. Some GC algorithms, including Epsilon GC, Shenandoah GC, and other experimental GCs, are disregarded due to a lack of availability and setup complexity.

3 EXPERIMENTAL METHODOLOGY

The subsequent part of this section provide details on the methodology used in this study, including information on the experimental settings, environments, and data used. Additionally, the availability of the code used in the experiments is also discussed.

3.1 Environments, Code and Data Availability

In this study, we conducted an analysis of various garbage collectors on the `Java version 20 (JDK 20)` platform with the runtime environment of `Java(TM) SE Runtime Environment (build 20+36-2344)` based on `Java HotSpot(TM) 64-Bit Server VM (build 20+36-2344, mixed mode, sharing)` running on `Windows Server 2019 Standard`. We used both fixed and varying heaps in garbage collection while performing the analysis on two benchmark suites, namely the `DaCapo Benchmark Suite`[1] [2] and the `Renaissance Benchmark Suite`[2] [8]. To ensure accuracy, we ran 60 iterations during the garbage collection procedures for the Renaissance Benchmark Suite. Finally, we used `GCeasy - A Universal GC Log Analyzer`[3], to analyze the performance data obtained from the experiments.

The code and data used in this study, as well as other related data found from the experiment such as logs, charts, files, and, graphics are available in this GitHub repository.[4][5]

3.2 Benchmark Suites

This study uses benchmarks from the DacCapo Benchmark Suite and the Renaissance Benchmark Suite. The benchmarks used in this study are provided below.

3.2.1 DacCapo Benchmark Suite. The following benchmarks make up the `DaCapo-9.12-bach` benchmark suite, which was released in 2009 are used in this study:

- **`avrora`** It simulates programs running on a grid of AVR microcontrollers.
- **`batik`** It produces Scalable Vector Graphics (SVG) images based on unit tests in Apache Batik.
- **`eclipse`** It executes non-gui performance tests for the Eclipse IDE.
- **`fop`** It generates a PDF file by parsing and formatting an XSL-FO file.

[1] https://www.dacapobench.org/
[2] https://renaissance.dev/
[3] https://gceasy.io/
[4] https://github.com/DataParadox/modern_gc/
[5] https://dataparadox.github.io/modern_gc/

- **h2** It executes a JDBCbench-like in-memory benchmark, running transactions against a model of a banking application.
- **jython** It interprets the pybench Python benchmark.
- **luindex** It uses lucene to index documents, such as the works of Shakespeare and the King James Bible.
- **lusearch** It uses lucene to perform a text search of keywords over a corpus of data, such as the works of Shakespeare and the King James Bible.
- **pmd** It analyzes Java classes for source code problems.
- **sunflow** It renders images using ray tracing.
- **tomcat** It runs queries against a Tomcat server and verifies resulting webpages.
- **xalan** It transforms XML documents into HTML, which simulates programs running on a grid of AVR microcontrollers.

3.2.2 Renaissance Benchmark Suite. The benchmarks used for this study as a part of the Renaissance Benchmark Suite include:

- **chi-square** It executes the chi-square test from Spark MLlib.
- **naive-bayes** It runs the multinomial Naive Bayes algorithm from the Spark ML library.
- **page-rank** It executes several PageRank iterations, using RDDs.
- **reactors** It runs benchmarks inspired by the Savina microbenchmark workloads in a sequence on Reactors.IO.
- **future-genetic** It runs a genetic algorithm using the Jenetics library and futures.
- **mnemonics** It solves the phone mnemonics problem using JDK streams.
- **philosophers** It solves a variant of the dining philosophers problem using ScalaSTM.
- **scala-kmeans** It executes the K-Means algorithm using Scala collections.
- **finagle-http** It sends many small Finagle HTTP requests to a Finagle HTTP server and awaits response.

3.3 Execution Procedures

To execute the garbage collection shell script, the following commands are used:

- **java** This command initiates the Java Virtual Machine (JVM) required to run the script.
- **-XX:+UnlockExperimentalVMOptions** This enables the use of experimental options that are not yet fully supported by the JVM.
- **-XX:+Use${name}GC** This specifies the use of the `${name}` Garbage Collector (GC) for garbage collection, where `${name}` is replaced by the name of the desired GC algorithm, for example, `-XX:+UseSerialGC`.
- **-Xmx${size}M** This sets the maximum heap size available to the JVM for object allocation to `${size}` megabytes.
- **-Xlog:gc*** This enables logging of GC-related events. The `gc*` argument specifies that all GC-related events should be logged. This option is particularly useful for analyzing GC behavior and identifying potential performance issues.

The Python script used to execute shell scripts on the JVM includes the following commands:

- **--b_suite** It specifies the benchmark suite used for evaluation, which can be either `dacapo` or `renaissance`.
- **--benchmark** It specifies the specific benchmark dataset used for evaluation.
- **--max_heap** It sets the maximum heap size available to the JVM, which must be in powers of 2 and greater than 512 MB.

- **`--varying_heap`** It specifies whether the heap size should be varied during the evaluation.

For example, for `xalan` benchmark dataset from `dacapo` benchmark suite with `max_heap = 2048` in `varying_heap` environment the GC Logs can generate using -

```
python performance_analysis.py --b_suite dacapo --benchmark xalan\
--max_heap 2048 --varying_heap True
```

And for `page-rank` benchmark dataset from `renaissance` benchmark suite with fixed heap environment the GC Logs can generate using -

```
python performance_analysis.py --b_suite renaissance\
--benchmark page-rank
```

Where, the `performance_analysis.py` file is provided in the GitHub repository mentioned in Section 3.1.

4 PERFORMANCE ANALYSIS

For analyzing the performance of different garbage collection algorithms in the JDK 20 environment, we utilized fixed and varying heap approaches. In the varying heap approach, we employed heap sizes of 512MB, 1024MB, 2048MB, and 4096MB. Our analysis includes various metrics such as -

- **Heap Size** The amount of memory allocated to the JVM for object allocation.
- **Throughput** The rate of processed transactions by the application in a given time.
- **Total GC Time** The total time taken by the garbage collector to free memory.
- **Avg GC Time** The average time taken by the garbage collector to free memory per GC cycle.
- **Minor GC Total Time** The total time taken by the young generation GC cycles.
- **Minor GC Avg Time** The average time taken by the young generation GC cycles per GC cycle.
- **Full GC Total Time** The total time taken by the full GC cycles.
- **Full GC Avg Time** The average time taken by the full GC cycles per GC cycle.
- **Pause Total Time** The total time the application stops for GC.
- **Pause Avg Time** The average time the application stops for GC per GC cycle.
- **Young Generation Allocation** The amount of memory allocated to the young generation of the heap.
- **Old Generation Allocation** The amount of memory allocated to the old generation of the heap.
- **Meta Space Allocation** The amount of memory allocated to the JVM's metadata space.
- **Total Concurrent Time** The total time taken by the concurrent garbage collector to free memory.
- **Avg Concurrent Time** The average time taken by the concurrent garbage collector to free memory per GC cycle.

4.1 Fixed Heap

4.1.1 DaCapo Benchmark Suite. We use all benchmarks from the DaCapo suite except `tradebenas` and `tradesoap` to evaluate the performance of Serial GC, Parallel GC, G1 GC, and ZGC. The comparison graphs for the DaCapo suite are given in Appendix A.1 and average metrics found from this experiment are given in "Table 1". Comparing the metrics for each garbage collector reveals that the ZGC algorithm has the maximum throughput among all the benchmarks. Additionally, it has the shortest total and average pause times. The ZGC algorithm also has the greatest concurrent

time total and concurrent time average. These metrics indicate that ZGC has the least impact on the execution time of the application.

In the majority of benchmarks, the Serial GC algorithm has the longest total GC time, full GC time, and young GC time. This indicates that the Serial algorithm spends a significant amount of time on garbage collection and has a significant impact on the execution time of the application. The Parallel GC algorithm has less total GC time, complete GC time, and young GC time than the Serial GC algorithm.

The G1 algorithm has the lowest allocation of the young generation for the majority of benchmarks. This indicates that the G1 algorithm manages the young generation heap effectively. However, it has a greater allocation of the old generation and meta-space than the other algorithms. This suggests that the G1 algorithm may not be as effective as other algorithms at managing the old generation heap and metaspace.

In terms of the examined performance metrics, the ZGC algorithm seems to perform the best among the provided garbage collection algorithms.

Table 1. Average performance values for DaCapo benchmark suite in fixed heap environment

Throughput	JVM Memory Size			Young GC Time		Full GC Time		Pause Time		Concurrent Time	
	Young	Old	Meta	Total	Avg	Total	Avg	Total	Avg	Total	Avg
80.358%	3.75 GB	1.33 GB	10.75 MB	6 s 510 ms	592 ms	1 sec 210 ms	605 ms	2 s 340 ms	213 ms	646 ms	646 ms

4.1.2 Renaissance Benchmark Suite. In the Renaissance benchmark suite, we employ nine benchmarks for the four selected garbage collectors illustrated in Appendix A.2; "Table 2" provides the average performance of these garbage collectors.

The ZGC algorithm has a 100% throughput for all benchmarks, indicating that it has no substantial impact on the execution time of the application. The ZGC algorithm has the lowest total pause time and average pause time, indicating that it has the lowest delay times compared to other algorithms.

The Serial GC algorithm has a lower throughput than other algorithms, as well as a higher total pause time and average pause time, which has a greater impact on the application's execution time. The full GC total time and average time are also greater for the Serial GC algorithm, indicating that the Serial GC algorithm requires more time than other algorithms to perform full GC.

The Parallel GC algorithm has a higher throughput than the Serial GC algorithm, as well as a lower total pause time and average pause time. The Parallel GC algorithm has a lower complete GC total time and average time than the Serial GC algorithm.

The Parallel GC algorithm and the G1 GC algorithm have comparable throughput, total pause time, and average pause time. However, the G1 GC algorithm has a longer complete GC total time and average time than the Parallel GC algorithm, indicating that it requires more time to perform full GC.

The ZGC algorithm has the highest throughput and the lowest pause times, whereas the Serial algorithm has the lowest throughput and the highest pause times. In terms of throughput and pause times, the Parallel and G1 GC algorithms have comparable performance, but the Parallel GC algorithm has faster complete GC times than the G1 GC algorithm.

Table 2. Average performance values for Renaissance benchmark suite in fixed heap environment

Throughput	JVM Memory Size			Young GC Time		Full GC Time		Pause Time		Concurrent Time	
	Young	Old	Meta	Total	Avg	Total	Avg	Total	Avg	Total	Avg
47.545%	11.05 GB	2.40 GB	129.44 MB	3 m 55 s 143 ms	28.20 ms	1 m 31 s 670 ms	129 ms	4 m 35 s 643 ms	81.80 ms	57 s 348 ms	98.50 ms

4.2 Varying Heap

4.2.1 DaCapo Benchmark Suite. We use five benchmarks from the DaCapo suite for heap-based performance analysis. Appendix B.1 illustrates performance as determined by various metrics, and "Table 3" depicts the average state of this experiment.

Comparing the values of the metrics found in each benchmark indicates that the performance metrics vary considerably depending on the GC algorithm, heap size, and benchmark applied. For example, in the `avrora` benchmark with a heap size of 512 MB, for instance, the ZGC algorithm has the highest throughput (100%), the lowest total GC time (33 ms), and the lowest pause time (33 ms), whereas the G1 algorithm has the lowest throughput (99.79%), the highest total GC time (20,000 ms), and the highest pause time (153.56 ms). The G1 algorithm has the lowest throughput (96.42%) and the highest total GC time (93,400 ms) in the `batik` benchmark with a heap size of 512 MB, while the ZGC algorithm has the greatest throughput (100%) and the lowest total GC time (0 ms). To achieve optimal performance, it is essential to select the appropriate GC algorithm and heap size based on the specific application requirements.

Table 3. Average performance values for DaCapo benchmark suite in varying heap environment

Throughput	JVM Memory Size			Young GC Time		Pause Time		Concurrent Time	
	Young	Old	Meta	Total	Avg	Total	Avg	Total	Avg
99.953%	153.56 MB	341.50 MB	11.94 MB	10 s 975 ms	323 ms	6.39 ms	1.60 ms	9 s 371 ms	247 ms

4.2.2 Renaissance Benchmark Suite. We use five benchmarks from the Renaissance benchmark suite to determine the performance of four garbage collection algorithms: Serial GC, Parallel GC, G1 GC, and ZGC in varying heap environments with heap sizes of 512, 1024, 2048, and 4096. In "Table 4" is a summary of the experiment, while Appendix B.2 depicts the performance based on various metrics. Based on the applied benchmark, the efficacy of each GC algorithm is inconsistent. For instance, the ZGC algorithm has the highest throughput for the `chi-square` benchmark, obtaining 99.99% for heap sizes of 2048 and 4096. G1 is the second-best algorithm in terms of throughput, attaining up to 93.07% for a 1024-element heap. For all heap sizes, the Parallel algorithm obtains a lower throughput than ZGC and G1. ZGC has the lowest values for total GC time and average GC time across all heap sizes.

G1 and ZGC have comparable throughput for the `naive-bayes` benchmark, attaining up to 98.46% and 98.02%, respectively. For all storage sizes, the Parallel algorithm has the lowest throughput. ZGC has the lowest total GC time and average GC time for all heap sizes, similar to the `chi-square` benchmark.

In terms of throughput and GC time, ZGC performs best overall for all benchmarks, followed by G1. In terms of throughput, the Parallel algorithm performs the worst and has the greatest total and average GC times. The heap size also influences the performance of the garbage collection (GC) algorithms, with larger heap sizes generally resulting in greater throughput and lengthier GC times.

Table 4. Average performance values for Renaissance benchmark suite in varying heap environment

Throughput	JVM Memory Size			Young GC Time		Full GC Time		Pause Time		Concurrent Time	
	Young	Old	Meta	Total	Avg	Total	Avg	Total	Avg	Total	Avg
36.606%	2.69 GB	2.59 GB	129.38 MB	21 m 53 s 656 ms	35.50 ms	15 m 29 s 827 ms	153 ms	27 m 47 s 183 ms	73.10 ms	14 m 53 s 741 ms	88.40 ms

5 CONCLUSIONS

We are evaluating the efficacy of garbage collectors available in JDK 20. Experiments are conducted with the DaCapo and Renaissance benchmark suites in both constant and variable heap environments. The results indicate that the ZGC algorithm has the maximum throughput and the lowest pause times among the examined garbage collectors. The Serial GC algorithm has the lowest throughput and the longest pause periods, and the G1 algorithm manages the old generation heap and metaspace less effectively. Therefore, our work provides developers with vital insights into the efficacy of garbage collection algorithms, which will assist them in selecting the optimal algorithm for their applications.

Our study focused on a subset of benchmarks from the DaCapo and Renaissance suites, with a heap-based experiment setup that was both fixed and variable. Due to time, complexity, and environmental constraints, we were forced to limit the number of benchmarks and GC algorithms we evaluated. Nonetheless, we were able to acquire vital insights into the performance of various GC algorithms under these particular conditions.

Looking forward, we acknowledge that there is still a lot of room for further exploration in this field. In the future, we intend to extend our investigation by analyzing a greater number of benchmark suites and garbage collection (GC) algorithms across a wider range of heap sizes. We also recognize the potential advantages of studying experimental GC algorithms as well as Epsilon GC and Shenandoah GC that are going to incorporate them into our research whenever feasible. In the future, we believe that our findings can contribute to the development of more efficient and effective garbage collection techniques, resulting in enhanced performance across a wide range of applications.

ACKNOWLEDGMENTS

We would like to express our gratitude to Dr. Rifat Shahriyar, Professor of the Department of Computer Science and Engineering at Bangladesh University of Engineering and Technology, for his invaluable guidance as a resource person during this study.

REFERENCES

[1] D. Beronić, N. Novosel, B. Mihaljević, and A. Radovan. 2022. Assessing Contemporary Automated Memory Management in Java – Garbage First, Shenandoah, and Z Garbage Collectors Comparison. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*. 1495–1500. https://doi.org/10.23919/MIPRO55190.2022.9803445

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. *SIGPLAN Not.* 41, 10 (oct 2006), 169–190. https://doi.org/10.1145/1167515.1167488

[3] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. 2015. A Performance Study of Java Garbage Collectors on Multicore Architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (San Francisco, California) *(PMAM '15)*. Association for Computing Machinery, New York, NY, USA, 20–29. https://doi.org/10.1145/2712386.2712404

[4] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*. 37–48.

[5] H. Grgic, B. Mihaljević, and A. Radovan. 2018. Comparison of garbage collectors in Java programming language. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1539–1544. https://doi.org/10.23919/MIPRO.2018.8400277

[6] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification.* Pearson Education.
[7] Java Platform. 2015. Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide.
[8] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019).* Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637
[9] P. Pufek, H. Grgić, and B. Mihaljević. 2019. Analysis of Garbage Collection Algorithms and Memory Management in Java. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* 1677–1682. https://doi.org/10.23919/MIPRO.2019.8756844
[10] Benjamin Zorn. 1990. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming.* 87–98.

A FIXED HEAP

A.1 DaCapo Benchmark Suite

The results found using the DaCapo benchmark suite for throughput comparisons are shown in "Figure 4", full and young GC average time in "Figure 5", average GC time, pause time, and concurrent time in "Figure 6", and JVM memory size allocated by each generation and meta space are given in "Figure 7".

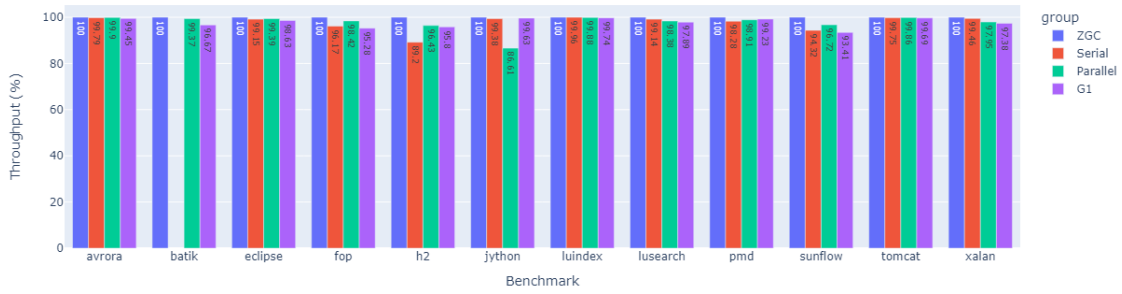


Fig. 4. Throughput comparisons in DaCapo suite for fixed heap.

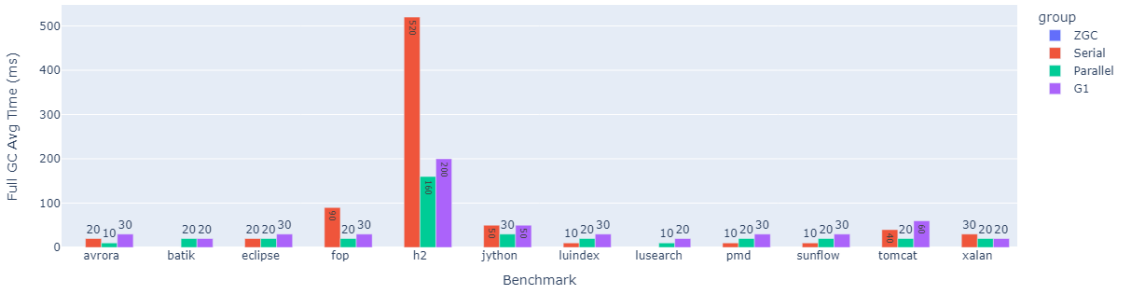
A.2 Renaissance Benchmark Suite

"Figure 8" shows the results obtained using the DaCapo benchmark suite for throughput comparisons, "Figure 9" illustrates the average full and young GC time, "Figure 10" demonstrates the average GC time, pause time, and concurrent time, and "Figure 11" displays the JVM memory size allocated by each generation and the meta space.

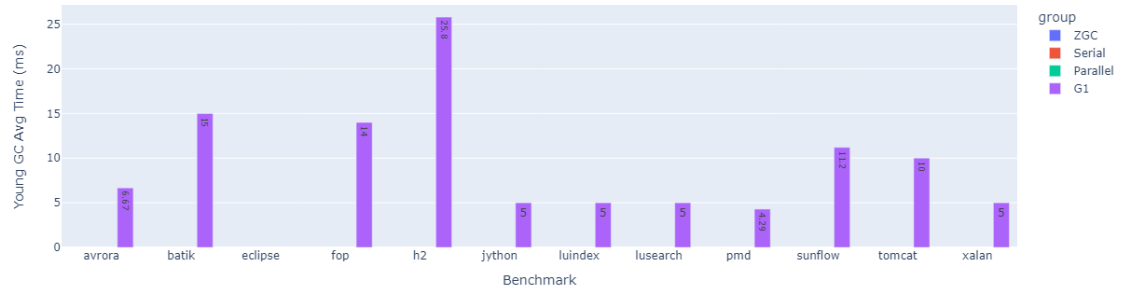
B VARYING HEAP

B.1 DaCapo Benchmark Suite

"Figure 12" depicts the results of throughput comparisons using the DaCapo benchmark suite; "Figure 13" depicts the full and minor GC average time; "Figure 14" depicts the average GC time, pause time, and concurrent time; and "Figure 15" depicts the JVM memory size allocated by each generation and meta space.



(a) Full GC average time



(b) Young GC average time

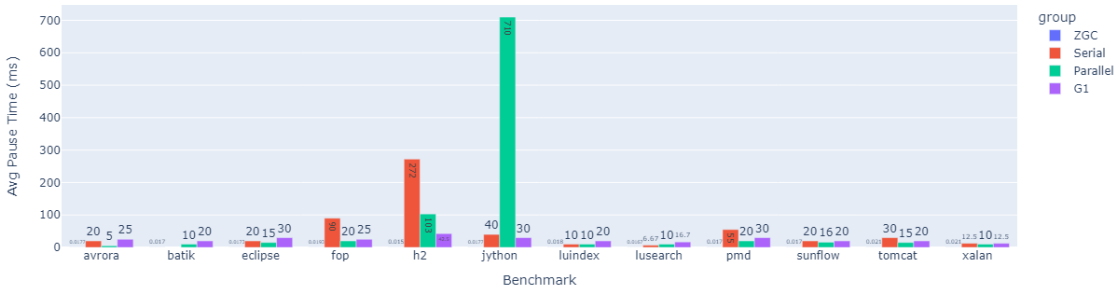
Fig. 5. Full and young GC average time in DaCapo suite for fixed heap.

B.2 Renaissance Benchmark Suite

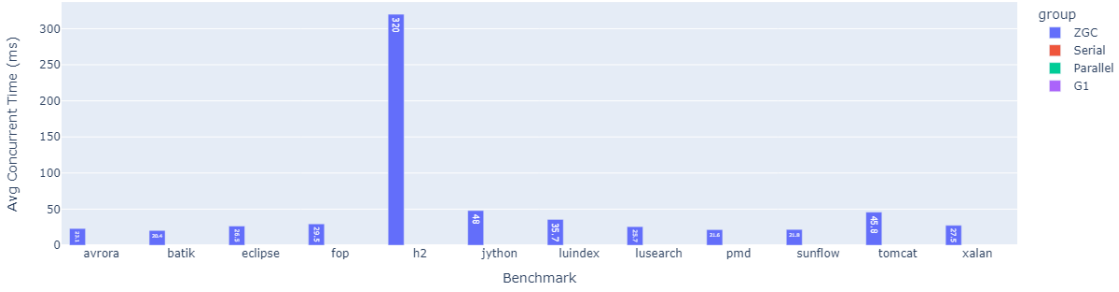
"Figure 16" displays the throughput comparison results from the DaCapo benchmark suite; "Figure 17" displays the average GC time for full and minor GCs; "Figure 18" displays the average GC time, pause time, and concurrent GC time; and "Figure 19" displays the memory size allocated by the JVM for each GC generation and meta space.

Received 14 April 2023

(a) Average GC time

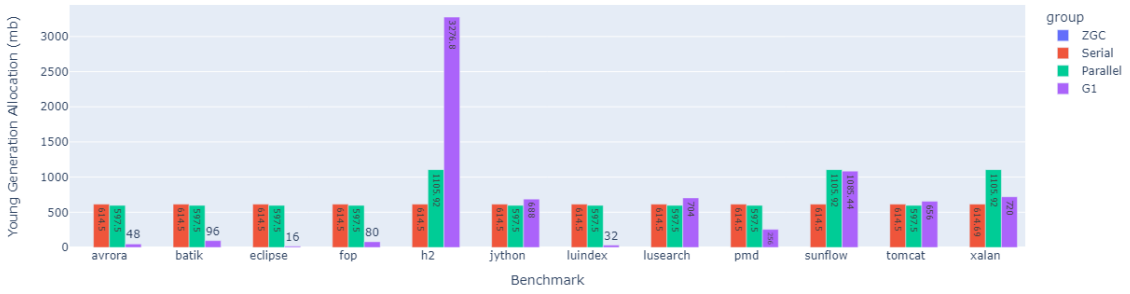


(b) Average pause time



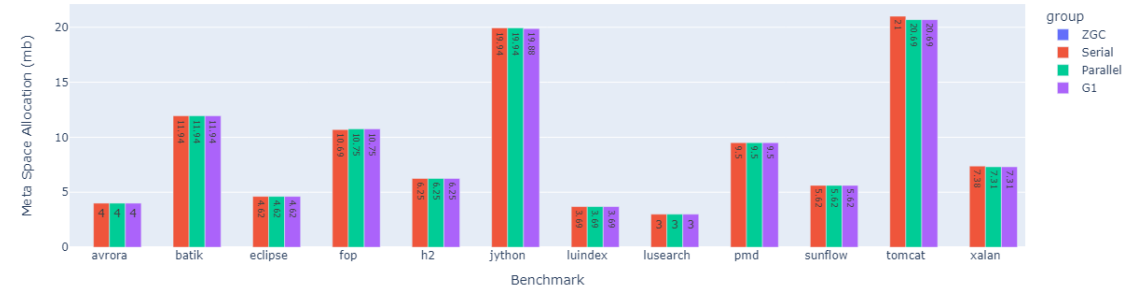
(c) Average concurrent time

Fig. 6. Average GC, pause, and concurrent time in DaCapo suite for fixed heap.



(a) Young generation allocation

(b) Old generation allocation



(c) Meta space allocation

Fig. 7. JVM memory allocations in DaCapo suite for fixed heap.

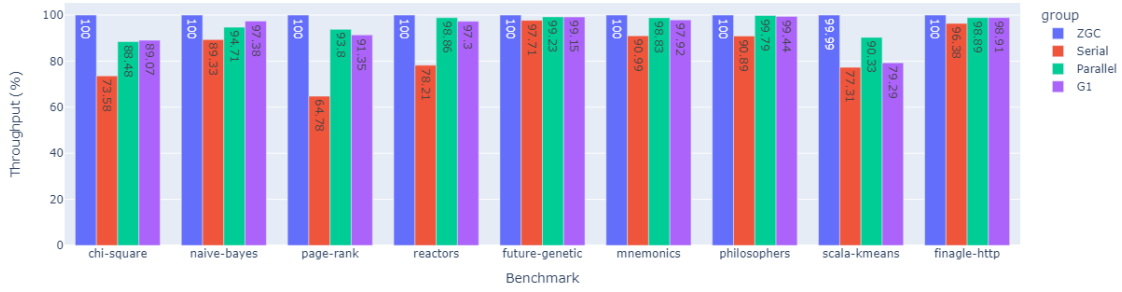
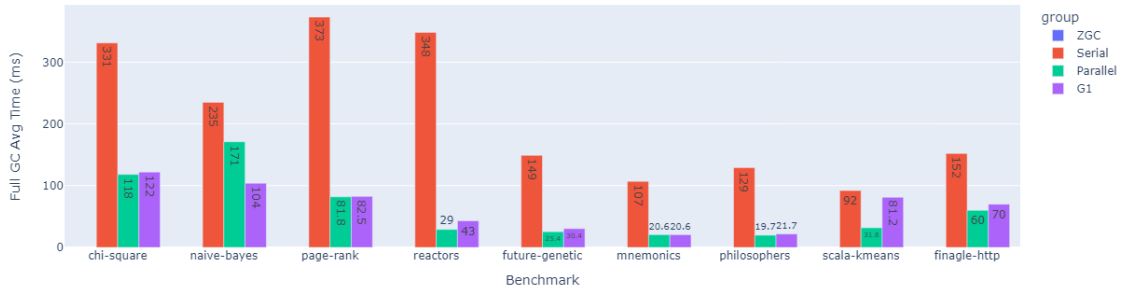
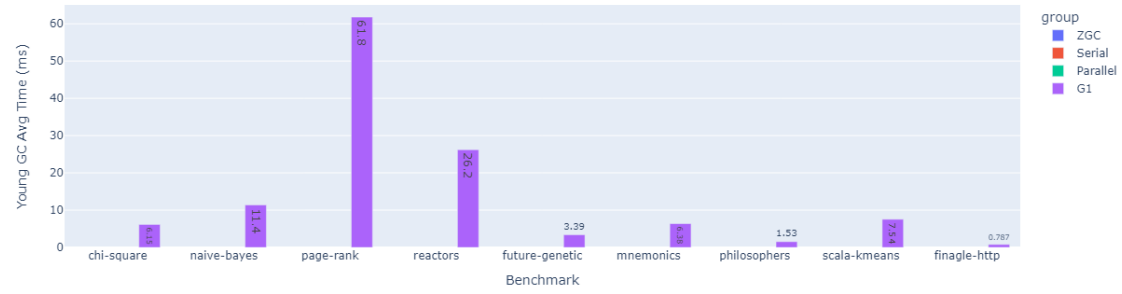


Fig. 8. Throughput comparisons in Renaissance suite for fixed heap.

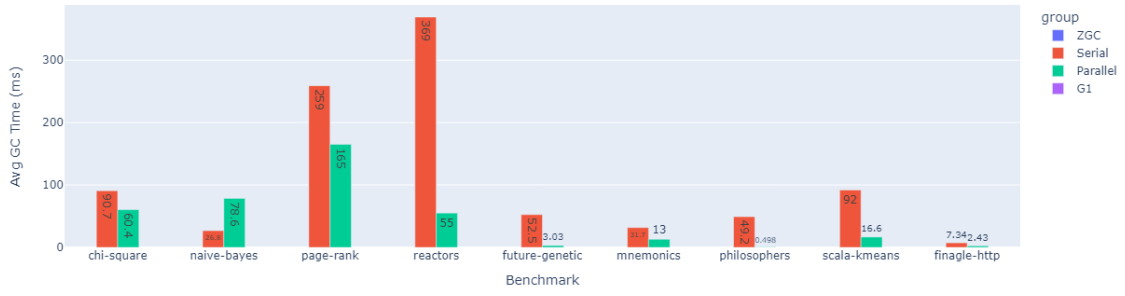


(a) Full GC average time

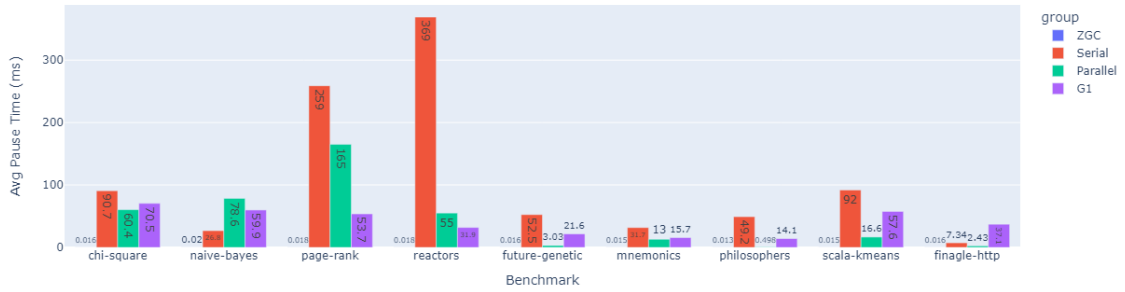


(b) Young GC average time

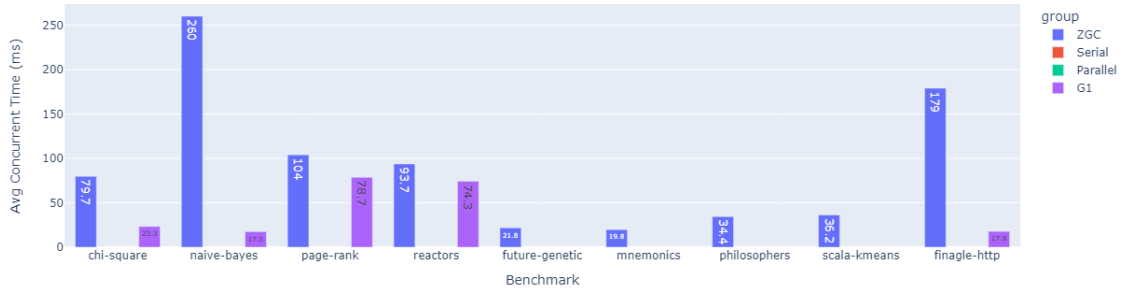
Fig. 9. Full and young GC average time in Renaissance suite for fixed heap.



(a) Average GC time

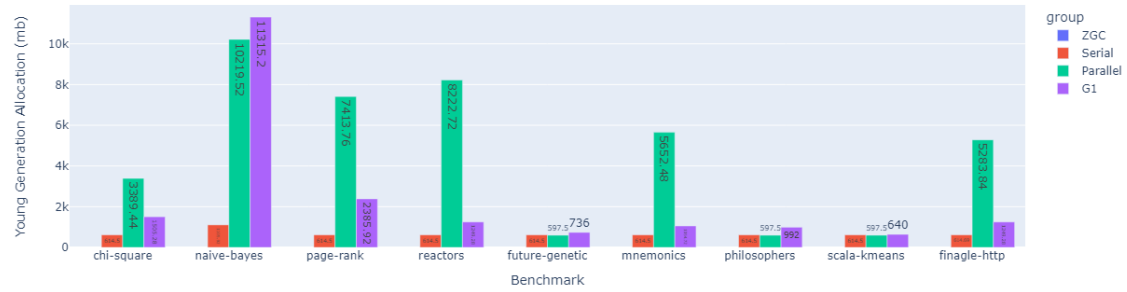


(b) Average pause time

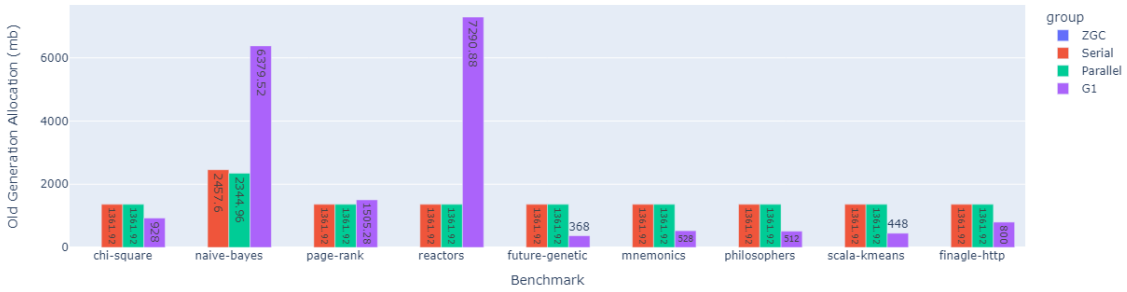


(c) Average concurrent time

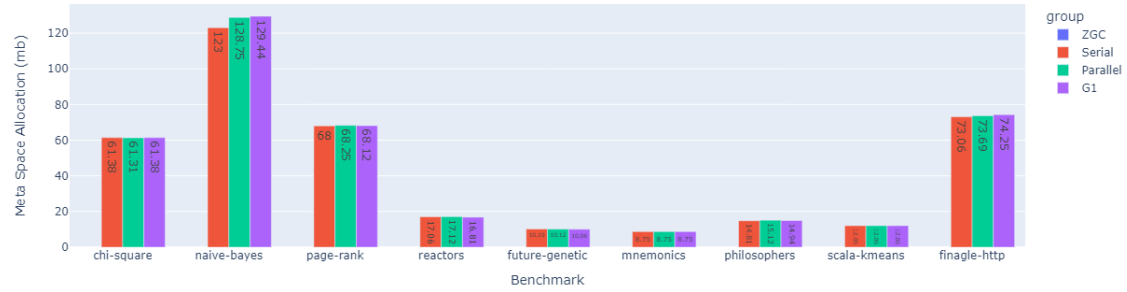
Fig. 10. Average GC, pause, and concurrent time in Renaissance suite for fixed heap.



(a) Young generation allocation



(b) Old generation allocation



(c) Meta space allocation

Fig. 11. JVM memory allocations in Renaissance suite for fixed heap.

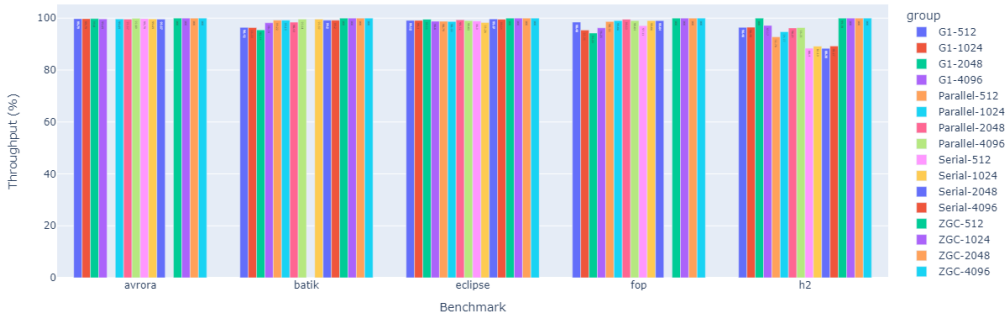
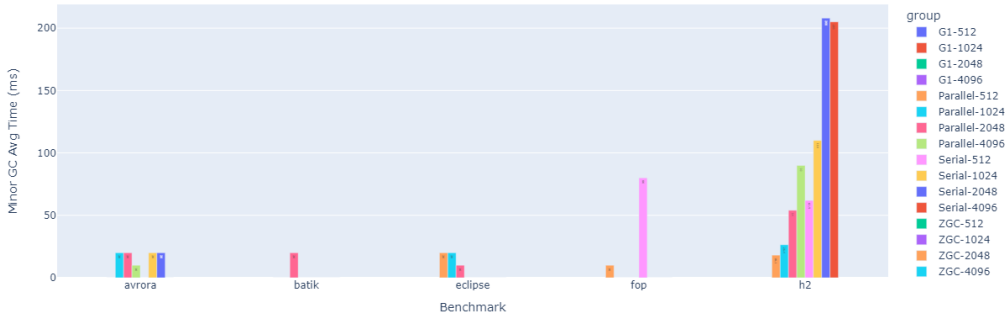


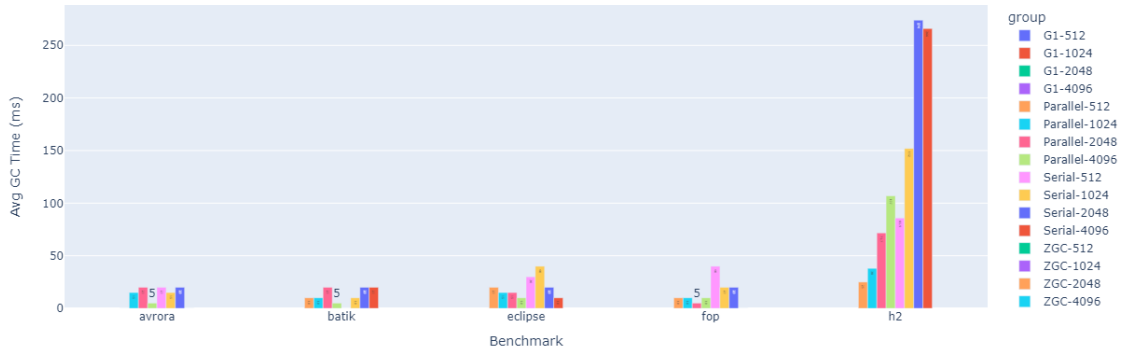
Fig. 12. Throughput comparisons in DaCapo suite for varying heap.

(a) Full GC average time

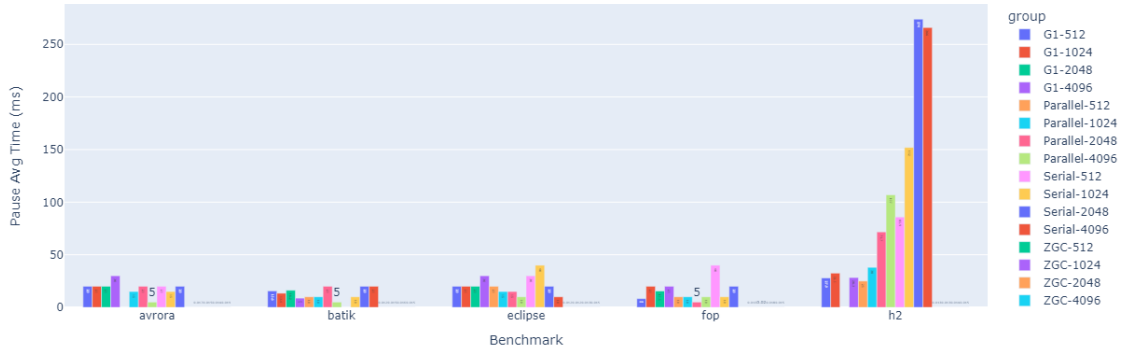


(b) Minor GC average time

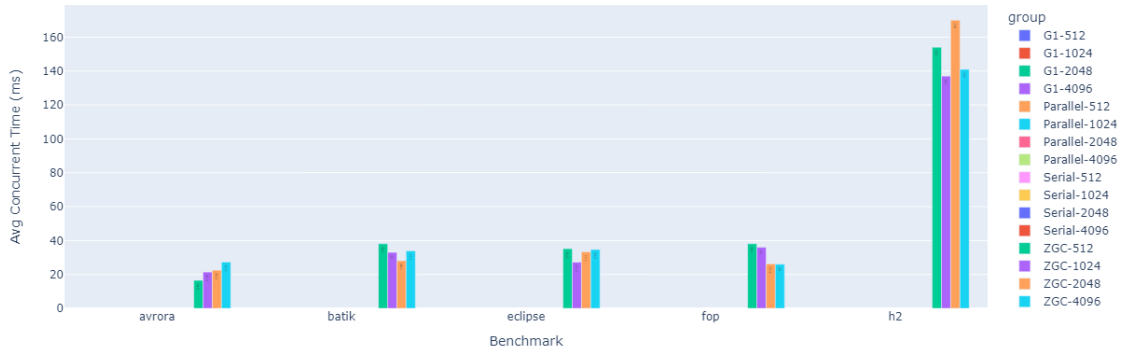
Fig. 13. Full and minor GC average time in DaCapo suite for varying heap.



(a) Average GC time

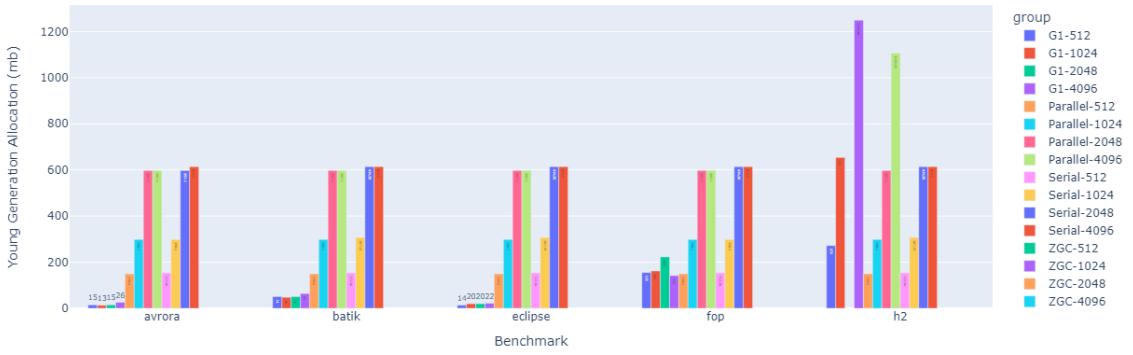


(b) Average pause time

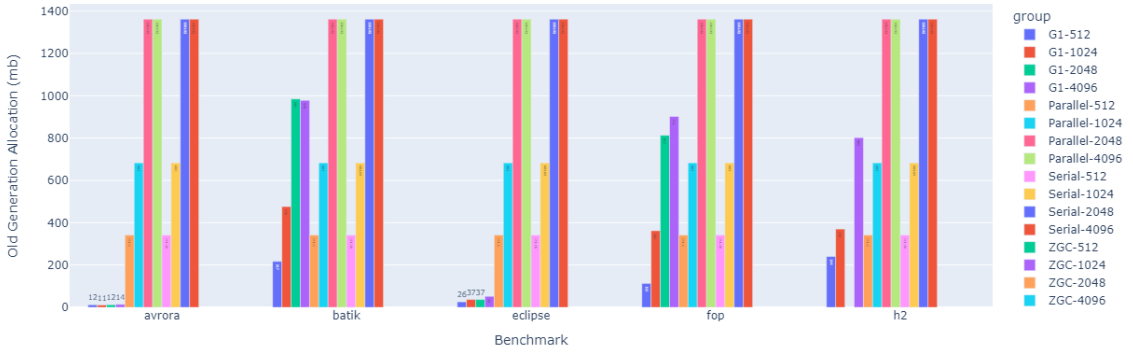


(c) Average concurrent time

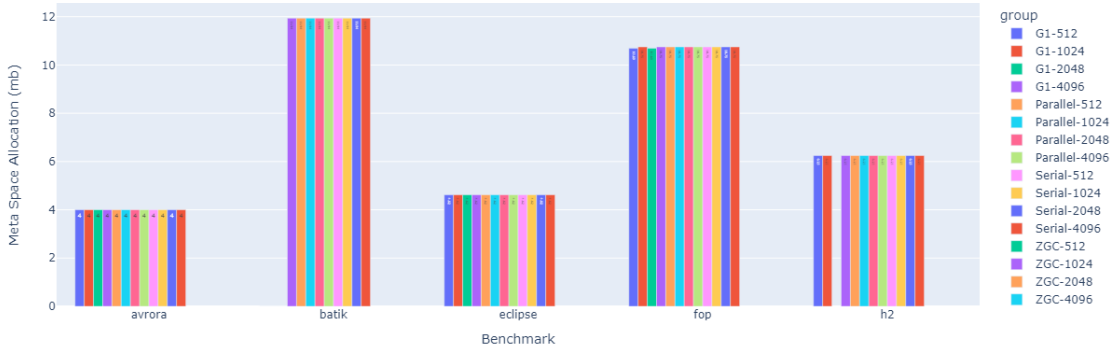
Fig. 14. Average GC, pause, and concurrent time in DaCapo suite for varying heap.



(a) Young generation allocation



(b) Old generation allocation



(c) Meta space allocation

Fig. 15. JVM memory allocations in DaCapo suite for varying heap.

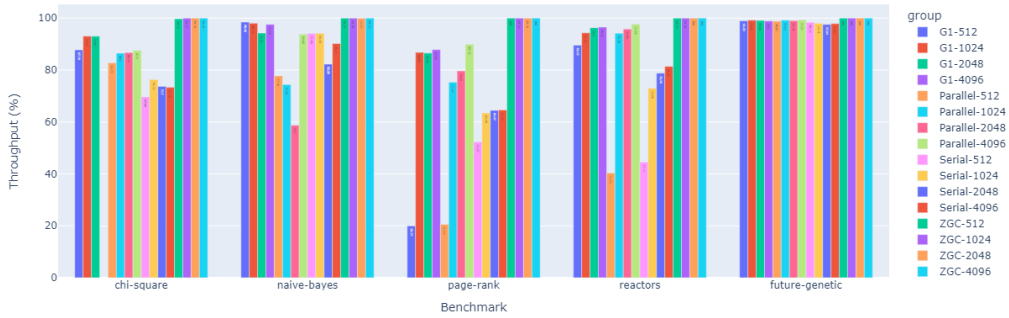
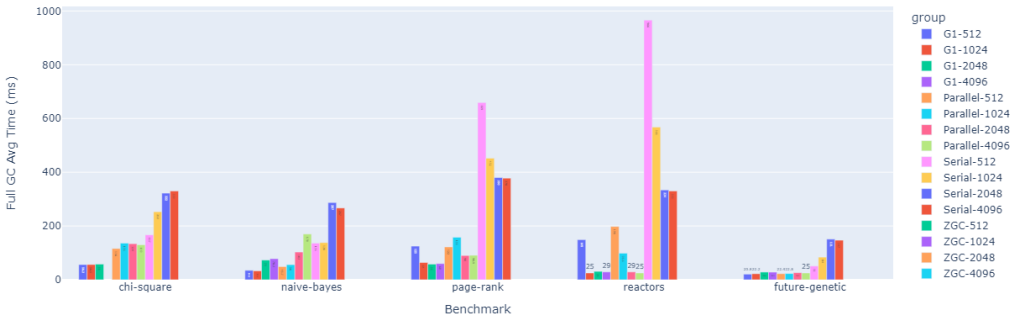
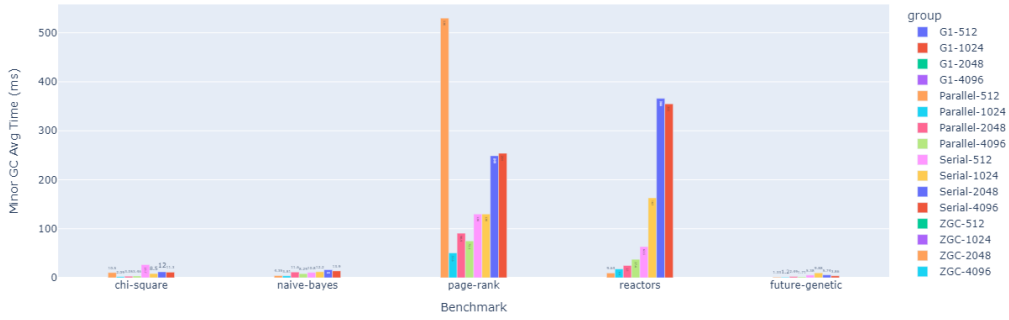


Fig. 16. Throughput comparisons in Renaissance suite for varying heap.

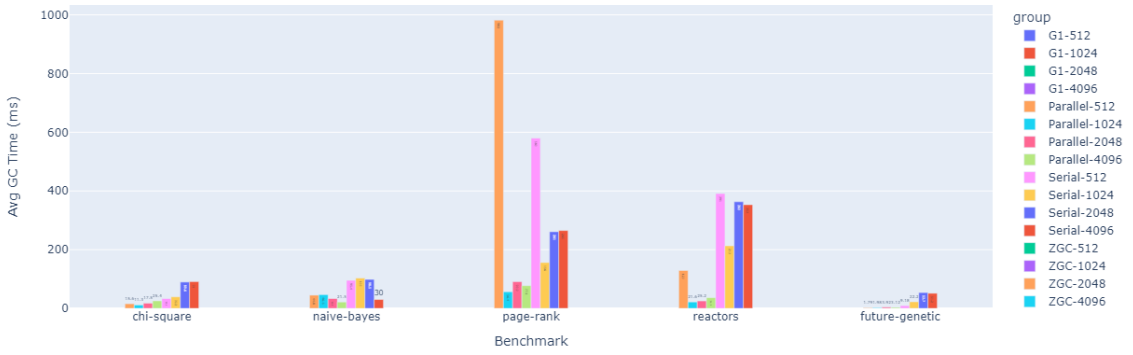


(a) Full GC average time

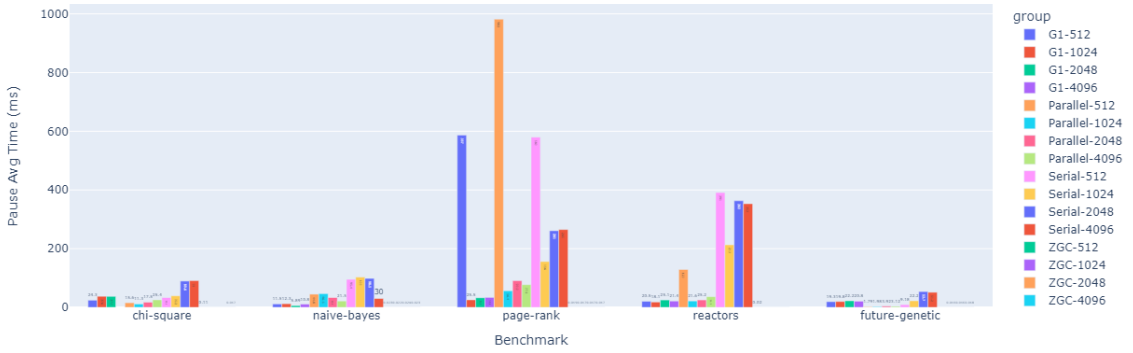


(b) Minor GC average time

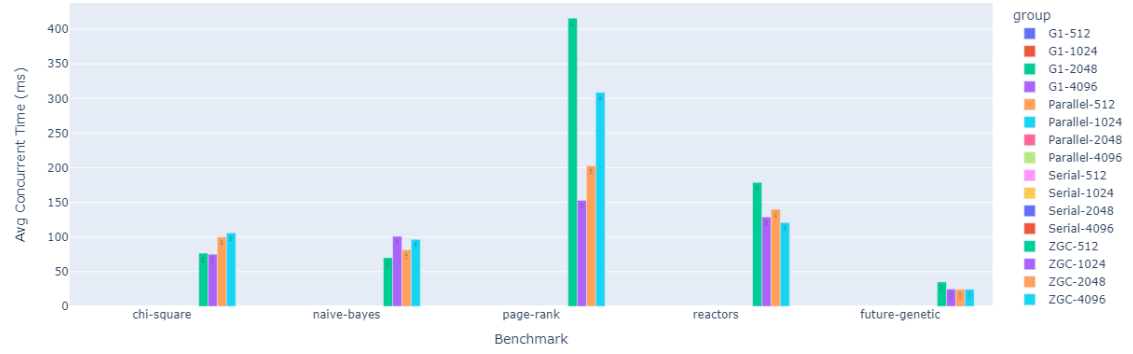
Fig. 17. Full and minor GC average time in Renaissance suite for varying heap.



(a) Average GC time

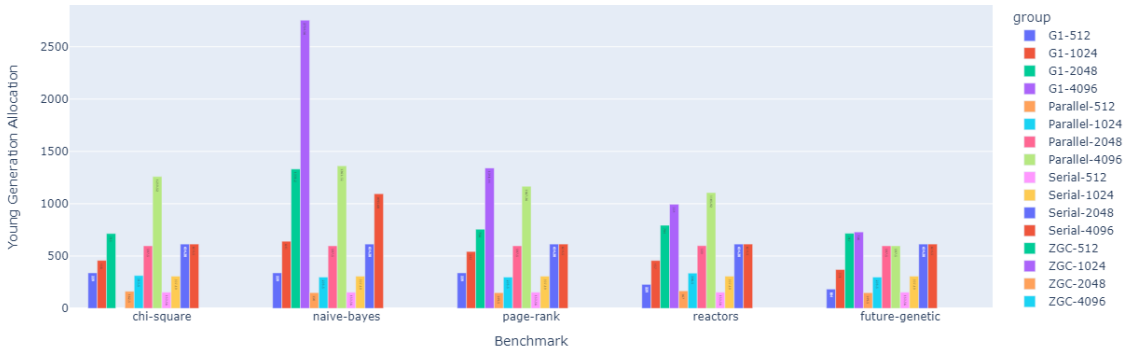


(b) Average pause time

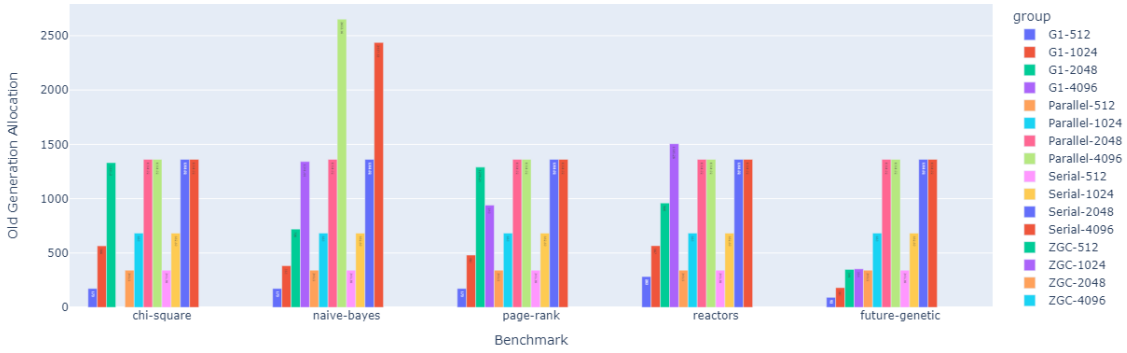


(c) Average concurrent time

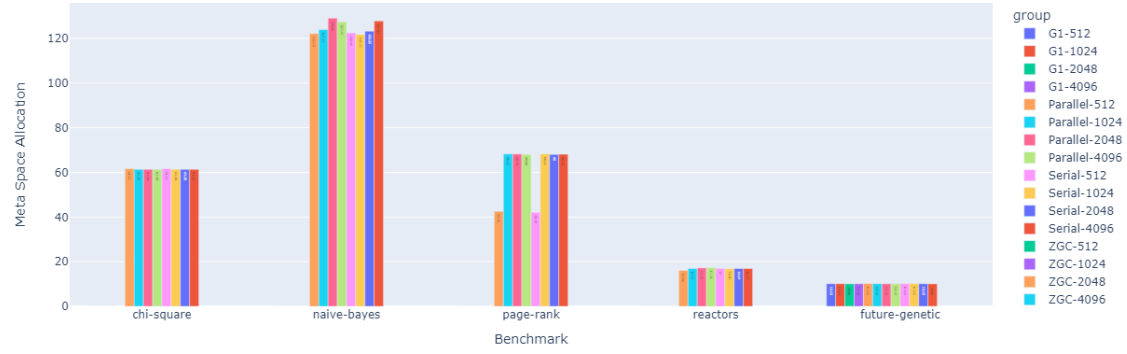
Fig. 18. Average GC, pause, and concurrent time in Renaissance suite for varying heap.



(a) Young generation allocation



(b) Old generation allocation



(c) Meta space allocation

Fig. 19. JVM memory allocations in Renaissance suite for varying heap.