Comparison of Garbage Collector Performance

The performance of garbage collectors (GCs) can significantly impact Java application performance, including throughput, latency, and memory usage. Here's a comparison of several commonly used garbage collectors in the JVM, focusing on their performance characteristics and suitability for different types of applications.

1. Serial Garbage Collector (SerialGC)

Characteristics:

- **Single-Threaded**: Uses a single thread for garbage collection.
- **Simple Design**: Easy to understand and implement.
- **Stop-The-World (STW)**: All application threads are paused during garbage collection.

Performance:

- **Throughput**: Generally lower due to frequent and potentially lengthy pauses.
- **Latency**: Can cause long pauses, making it unsuitable for latency-sensitive applications.
- **Best For**: Small applications with limited memory requirements or single-threaded applications where simplicity is preferred.

2. Parallel Garbage Collector (ParallelGC)

Characteristics:

- **Multi-Threaded**: Utilizes multiple threads to perform garbage collection.
- **Throughput-Oriented**: Focuses on maximizing application throughput by reducing the proportion of time spent in garbage collection.

Performance:

- **Throughput**: High, as it minimizes the time spent in GC relative to the time spent executing application code.
- **Latency**: Can cause longer pauses compared to other collectors, especially with large heaps.
- **Best For**: Applications with large datasets or batch processing needs where high throughput is more critical than low latency.

3. G1 Garbage Collector (G1GC)

Characteristics:

- **Region-Based**: Divides the heap into small regions and collects garbage in a region-by-region manner.
- **Balanced Approach**: Aims to balance throughput and pause times, with user-defined pause time goals.

Performance:

- **Throughput**: Generally lower than ParallelGC due to overhead from managing regions and pause time goals.
- **Latency**: More predictable and typically lower pause times compared to ParallelGC. Allows setting pause time goals with `-XX:MaxGCPauseMillis`.
- **Best For**: Large heaps where predictable pause times are needed and some reduction in throughput can be tolerated.

4. Z Garbage Collector (ZGC)

Characteristics:

- **Low-Latency**: Designed to provide very low pause times, even with large heaps.
- **Concurrent Collection**: Performs most garbage collection operations concurrently with the application.

Performance:

- **Throughput**: Lower compared to ParallelGC due to additional overhead of concurrent collection.
- **Latency**: Extremely low pause times, usually in the range of milliseconds, suitable for latency-sensitive applications.
- **Best For**: Applications requiring minimal pause times, particularly with large heaps (e.g., large-scale data processing or real-time systems).

5. Shenandoah Garbage Collector

Characteristics:

- **Low-Latency**: Similar to ZGC, aimed at minimizing pause times.

- **Concurrent Collection**: Most of the GC work is done concurrently with application threads.

Performance:

- **Throughput**: Similar to ZGC, with overhead from concurrent collection potentially impacting throughput.
- **Latency**: Consistently low pause times, with improvements over older collectors in managing heap fragmentation and reducing GC pause times.
- **Best For**: Applications where low latency is critical and large heaps are involved.

Choosing the Right Garbage Collector

- **For High Throughput**: Use **ParallelGC** for applications where throughput is critical, and latency can be tolerated.
- **For Predictable Latency**: **G1GC** offers a balance between pause times and throughput, with adjustable pause time goals.
- **For Minimal Latency**: **ZGC** and **Shenandoah** are ideal for applications requiring very low pause times, particularly with large heaps.

Choosing the appropriate garbage collector involves understanding your application's requirements for latency, throughput, and heap size, and tuning the garbage collector settings accordingly. Each GC offers trade-offs, and the best choice depends on specific performance goals and workload characteristics.