Liberica NIK
Selecting the Garbage Collector

Liberica NIK
Revision 1.0
October 17, 2023

bell/soft

Contents

1. Introducing Garbage Collection

Garbage collection (GC) is an important and inevitable part of JVM that affects the overall performance of an application. It is beneficial to know which Garbage Collector (GC) is used by the JVM running in your Liberica NIK environment.

Liberica NIK provides three Garbage Collectors: **Serial**, **Parallel**, and **Epsilon** (no op).

- Serial GC is a simple, single-threaded GC algorithm that performs garbage collection in a single thread one by one. It is suitable for small applications or systems with low memory requirements.
- Parallel GC uses multiple threads to speed up garbage collection, making it more efficient for large applications running in a multiprocessor environment.
- Epsilon GC is a no-operation (no-op) GC that does not collect any garbage. It only handles the allocation of memory. Once the available Java heap is exhausted, the JVM shuts down.

The information in this document is accurate as of NIK version 24.0.0.

It is important to note that Serial and Parallel GCs are production-ready in Liberica NIK 24.0.0.

2. GC options in NIK/Graal

One feature specific to AOT-compiled native images is that you can specify different options that apply at two different stages of building and running native images. One stage is the so-called “image build time” when Java bytecode is compiled into native code. The other is the “image run-time” when the resulting image is executed.

Likewise, the options that control GC behavior fall into two categories. Some are applied at the image build time, and are baked into the image. Others can be specified during the native image invocation.

You can obtain a complete list of `native-image` options by running the following command:

```
native-image --expert-options-all
```

Image build time options

These options are passed to native-image using `-H:±Flag` or `-H:Setting=<value>` syntax.

The following options apply to any GC during the image build time:

Option	Description
`-H:AlignedHeapChunkSize`	The size of an aligned chunk.
`-H:HeapChunkHeaderPadding`	The Number of bytes at the beginning of each heap chunk that are not used for payload data, that is bytes that can be freely used as metadata by the heap chunk provider.
`-H:LargeArrayThreshold`	The size at or above which an array is allocated in its own unaligned chunk.
`-H:±TreatRuntimeCodeInfoReferencesAsWeak`	Determines if references from runtime-compiled code to Java heap objects should be treated as strong or weak.
`-H:±VerifyHeap`	Verify the heap before and after each collection.
`-H:±ZapChunks`	Fill unused memory chunks with a sentinel value.
`-H:±ZapConsumedHeapChunks`	After use Fill memory chunks with a sentinel value.
`-H:±ZapProducedHeapChunks`	Before use Fill memory chunks with a sentinel value.

The following options apply only to Serial and Parallel GC during the image build time:

Option	Description
`-H:±CountWriteBarriers`	Instrument write barriers with counters.
`-H:±GreyToBlackObjRefDemographics`	Develop demographics of the object references visited.
`-H:±IgnoreMaxHeapSizeWhileInVMOperation`	Ignore the maximum heap size while
`-H:±ImageHeapCardMarking`	Enables card marking for image heap objects, which arranges them in chunks. Automatically enabled when supported.
`-H:InitialCollectionPolicy`	The garbage collection policy, either Adaptive (default) or BySpaceAndTime.
`-H:MaxSurvivorSpaces`	Maximum number of survivor spaces.
`-H:SoftRefLRUPolicyMSPerMB`	This number of milliseconds multiplied by the free heap memory in MByte is the time span for which a soft reference will keep its referent alive after its last access.
`-H:±VerifyAfterGC`	Verify the heap after doing a garbage collection if VerifyHeap is enabled.
`-H:±VerifyBeforeGC`	Verify the heap before doing a garbage collection if VerifyHeap is enabled.
`-H:±VerifyReferences`	Verify all object references if VerifyHeap is enabled.
`-H:±VerifyReferencesPointIntoValidChunk`	Verify that object references point into valid heap chunks if VerifyHeap is enabled.

Option	Description
`-H:±VerifyRememberedSet`	Verify the remembered set if VerifyHeap is enabled.
`-H:±VerifyWriteBarriers`	Verify write barriers.

Run-time options

Run-time options are passed to the binary created by the `native-image` tool using regular Java syntax: `-XX:±Flag` or `-XX:Setting=<value>`. In addition, you can supply them to the `native-image` invocation using `-R:±Flag` or `-R:Setting=<value>` (note the difference from image build time options which use `-H:`). The resulting binary is compiled with the default values you provided.

The following options apply to any GC during the image startup:

Option	Description
`-H:±DisableExplicitGC`	Ignore calls to `System.gc()`.
`-H:±ExitOnOutOfMemoryError`	Exit on the first occurrence of an out-of-memory error that is thrown because the Java heap is out of memory.
`-H:MaxHeapSize`	The maximum heap size at run-time, in bytes.
`-H:MaxNewSize`	The maximum size of the young generation at run-time, in bytes
`-H:MaximumHeapSizePercent`	The maximum heap size as percentage of physical memory.
`-H:MaximumYoungGenerationSizePercent`	The maximum size of the young generation as a percentage of the maximum heap size.
`-H:MinHeapSize`	The minimum heap size at run-time, in bytes.
`-H:±PrintGC`	Print summary GC information after each collection.
`-H:ReservedAddressSpaceSize`	The number of bytes that should be reserved for the heap address space.
`-H:±VerboseGC`	Print more information about the heap before and after each collection.

The following options apply only to Serial and Parallel GC during the image startup:

Option	Description
`-H:PercentTimeInIncrementalCollection`	Percentage of total collection time that should be spent on young generation collections if the collection policy `BySpaceAndTime` is used.
`-H:MaxHeapFree`	The maximum free bytes reserved for allocations, in bytes (0 for automatic according to GC policy).
`-H:±TraceHeapChunks`	Trace heap chunks during collections if `+VerboseGC` is set.
`-H:±CollectYoungGenerationSeparately`	Determines if a full GC collects the young generation separately or together with the old generation.
`-H:±PrintGCSummary`	Print summary GC information after application main method returns.
`-H:±PrintGCTimes`	Print the time for each of the phases of each collection if `+VerboseGC` is set.

One additional option applies exclusively to Parallel GC during the image startup:

Option	Description
`ParallelGCThreads`	Number of GC worker threads.

3. Monitoring memory usage with JMX/JFR metrics

Native images support `MemoryMXBean` and `MemoryPoolMXBean` JMX interfaces. This article explains how they can be used for monitoring memory.

The following JFR events are supported:

- jdk.AllocationRequiringGC
- jdk.GCHeapSummary
- jdk.GCPhasePause
- jdk.GCPhasePauseLevel
- jdk.GarbageCollection
- jdk.ObjectAllocationInNewTLAB
- jdk.ObjectAllocationOutsideTLAB
- jdk.ObjectCount
- jdk.ObjectCountAfterGC
- jdk.SystemGC

4. Getting and analyzing GC logs

The following options help you get and analyze GC logs.

Before you start analyzing GC logs, you can check the type of GC used in your system by running the `-Xlog:gc=info` command.

- `-XX:+PrintGC` - Produces logging output similar to Java HotSpot when used at image run-time.
- `-XX:+VerboseGC` - Provides more verbose output showing how each GC round has affected the heap.
- `XX:+PrintGCTimes` - Additionaly shows how long is each GC work phase.

5. Selecting proper GC for the application

Important:

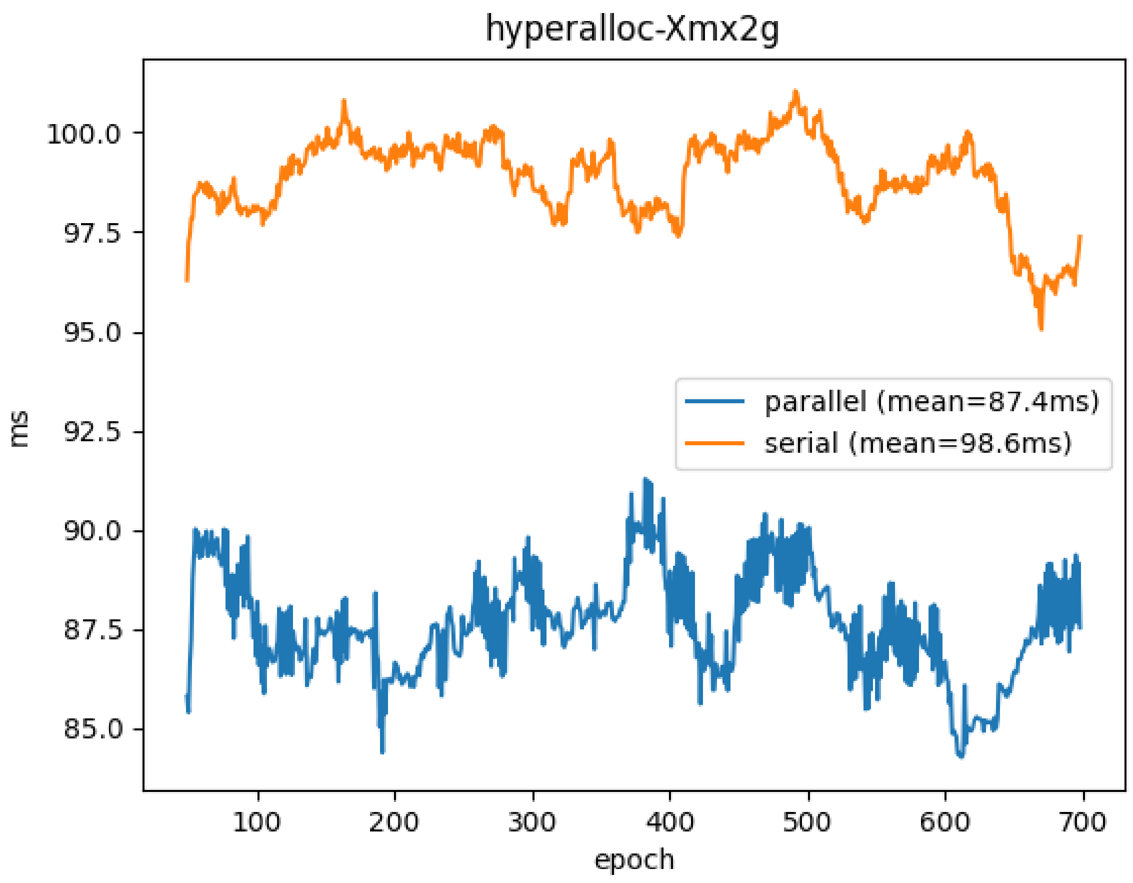
Serial and Parallel GCs are production-ready in Liberica NIK 24.0.0.

Serial and Parallel GC performance

Lower pause times have effect on application performance. If a particular service needs to have the least possible response times, Serial GC is not your choice.

The Parallel GC implementation extends a variety of GC types available in Liberica NIK and allows to improve response time.

Corresponding pause results were measured by natively compiling and running HyperAlloc benchmark from Heapothesys project. Numbers in the chart below are GC pause times in milliseconds. The benchmark was executed on Ubuntu, 8-core i7 CPU with 8 worker threads and incremental collection turned off.



For more information, see [Parallel garbage collector](#).

Choosing the GC

Serial GC is the oldest garbage collection mechanism existing from the early days of Java and has minimal overhead. It is suitable for memory and CPU constraint devices, but there can be long pauses in application work, especially if a significant amount of memory is involved.

If you have a large application running on a multicore system, we recommend using Parallel GC because it has more predictable pause time while achieving higher throughput and may shorten GC pauses.

Epsilon GC is useful for measuring program startup time, because it removes fluctuations related to memory management.

Liberica NIK

Selecting the Garbage Collector

bell/soft