G1GC Feedback

Introduction

As described in JEP 248, in JDK 9 the default garbage collector will switch from Parallel Garbage Collector (Parallel GC) to G1 Garbage Collector (G1GC).

This wiki-page aims to outline the basic JVM parameters switching to G1GC, and how you can help collecting data comparing the G1GC and Parallel GC.

Usually we would like to ask for gc logs, printed with **'**-XX:+PrintGCTimeStamps -XX:+PrintGCDetails**'.** Some time to diagnose further, we might ask for more detailed information (please see 'GC Parameters to provide more information'). We would like to focus on minimally tuned G1GC, either by not setting any flags or just setting the minimum and maximum heap size for you application and potentially a pause target for G1GC. The gc logs can be sent to public mailing list, or if you want to send privately, please contact us. Sometimes, a Java Flight Recorder recording is helpful as well.

Basic GC Parameters:

-Xms<min_heap_size> -Xmx<max_heap_size> -XX:MaxGCPauseMillis=<n> -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Xloggc:<gcfile>

For performance purpose, we usually set -Xms and -Xmx the same. But it is ok to set them the different value. If not set, the JVM will set those ergonomically.

MaxGCPauseMillis is important for G1GC. It determines the young gen size. The default is 200ms.

PrintGCTimeStamps is important for gc log analysis.

PrintGCDetails provides how pause time is distributed among gc steps.

loggc provides a way to write gc logs to a separate file, instead of stout. We recommend to do so.

GC Parameters to provide more information

If you experience any bad behavior (or if things run well) we might need further data to be able to do a more in-depth analysis to determine the cause of any issue. These flags might impact performance due to the fact that they collect more and extensive low-level data.

-XX:+PrintAdaptiveSizePolicy

This flag provides information about how g1 Ergonomics decides young gen size, when to start marking cycle, etc.

-XX:+PrintHeapAtGC

Print heap layout before and after gc.

-XX:+PrintTenuringDistribution

Print the age distribution among the objects in young gen.

-XX:+UnlockExperimentalVMOptions -XX:G1LogLevel=finest

This flag prints more detailed information, for example, pause time per gc thread.

-XX:+UnlockDiagnosticVMOptions -XX:+G1PrintRegionLivenessInfo

This flag prints liveness of each region before and after sorting at the end of marking cycle.

-XX:+UnlockDiagnosticVMOptions -XX:G1SummarizeRSetStats -XX:G1SummarizeRSetStatsPeriod=<n>

This flag prints out remember set statistics, every n gc cycles.

-XX:+UnlockDiagnosticVMOptions -XX:+G1SummarizeConcMark

This flag summarize concurrent marking information.

-XX:+PrintGCApplicationStoppedTime -XX:+PrintGCApplicationConcurrentTime

Print time when application is stopped due to safe points, and the time the application executes in between.

For how to understand the gc logs, please refer to blogs about how to print and understand basik G1GC logs.

For how to understand the printGCAdaptiveSizePolicy, please refer to blogs about how to understand G1GC ergonomics.

What to collect

We would like to get your feedback on all experiences, good, neutral and bad. Usually bad experiences is what generates blogs and emails, the other data points are not always easy to get.

Metrics
The following we see as the key metrics to gather data around to determine the impact of the switch. Not all metrics will be interesting for every application, but a default collector needs to behave well in them as a whole.

- **Throughput** - Code Performance
  - The pure application performance should not be too adversely affected by which GC is used.
  - Due to the heavier write barriers used in G1 there may be a performance impact. But the impact varies for different applications.
  - **Goal:** The performance is good enough so the general recommendation does not end up being to set the Parallel GC as part of application launch scripts.

- **Startup Time** - VM Initialization
  - G1 is a more complex GC than Parallel GC and requires a bit more infrastructure to be configured to enable concurrent marking and mixed collections.
  - This is most critical for short lived applications (such as simple command line tools or build tools), where the execution time is largely impacted by the time it takes to initialize the VM.
  - **Goal:** The VM initialization time must not have a major impact the run time of short running applications
- **Footprint** - Total Memory Usage
  - G1 is a more complex GC than Parallel GC and requires a bit more infrastructure to be configured to enable concurrent marking and mixed collections.
  - This also ties into the Ergonomics metric below in that the Ergonomics need to keep the memory usage balanced by not expanding the heap too aggressively.
  - **Goal**: The total memory usage of the Java process must be kept reasonable.
- **Ergonomics** - GC Behavior and Pauses
  - A default GC need to be able to handle a reasonably large set of different types of applications Out-Of-The-Box without extensive tuning.
    - A user should be able to just run their Java application, potentially just setting the heap size and pause time goal, and have it behave well.
    - The GC needs to be able size the heap and its sub components using dynamic ergonomics.
  - The default GC needs to be able to avoid long costly Full collections by using Ergonomics to self tune depending on the workload.
  - **Goal**: The GC Ergonomics must be able to handle common applications without causing performance overhead or Full GCs

Applications
We are focusing on general Java applications, and not on applications that are very dependent on extreme GC tuning. Applications not considered a target for this switch includes applications requiring low latency with milliseconds, micro or no-pause requirements, nor very large heap applications that use multiple of 10s or 100s of GB live data. The focus is on applications that are normally not tuned, either because there has never been a need to do so or they are used by people not accustomed to GC tuning.

- **Unaffected Applications**
  - Client VM Application: The Client VM will continue to use the Serial Collector, and applications such as Applets and regular Java applications using the 32bit JRE on Windows with only the client VM would not be affected.

- **Tuned Applications:** Any application that specifically sets the GC will not see any effect of this change.

- **Affected Applications**
  - Command Line Tool: Command line tools are often batch type jobs and are often fairly short running, so the GC pauses are of less concern as long as the execution is finished as soon as possible. Example tools are the Java Compiler, Scala Compiler, Maven and Gradle. The key metrics are Startup Time and Ergonomics to make sure the tool starts fast and that the GC can quickly adapt to the behavior of the application. For semi long running compilations the Throughput will become a factor as well.
  - UI Application: UI applications often don't select a specific GC, they might tune the heap size depending on the requirements of the application. End users of UI:s most likely don't know how to tune the VM, or the fact that the application is written in Java. As an example NetBeans by default only increases the minimum heap size as the only GC related tuning and let the ergonomics handle the rest. As a data point here, Java Mission Control is configured to run with G1 without issues so far. For most desktop UI application pure throughput is less critical. What tends to most noticeable is stalls due to GC pauses, so Ergonomics is probably the most important metric, together with Footprint as you do not want you application to use up all the memory on you desktop/laptop.
  - Applications using the Default GC:
    This is the area where we need the most help. We have internal workloads and benchmarks we can run, but that set will always be infinitely small compared to what is used across the Java community. This is also the reason why we enable this early to expand the amount of testing and feedback we are able to get.

For these applications all the metrics apply, but their importance vary from workload to workload.

Java Flight Recorder

Java Flight Recorder(JFR) is a commercial feature that provides java process profiles. Here is a link to Oracle document about JFR.

Here is another helpful blog.

Where to ask questions and share data

Please send your questions and data to hotspot-gc-use@openjdk.java.net or hotspot-gc-dev@openjdk.java.net.

If you prefer to send your log in private, you can send emails to yu.zhang@oracle.com. We will find a way that works for both parties.

Where can I see what has been collected and analyzed

We will create sub-pages to this wikipage where you will be able to find different workloads, their performance and behavior with G1.

Where to get JDK 9 early access build with G1GC as default

from JDK 9 Out Reach Page

- JDK 9 Early Access builds may also be provided by third parties. Oracle publishes regular JDK 9 builds at http://jdk9.java.net, for example.
- You can build JDK 9 yourself by following the build instructions to build the OpenJDK source code in the jdk9/jdk9 forest. Please check the list of supported build platforms to verify that your build platform is supported before you attempt building JDK 9 from source yourself.