Alpaquita Linux
Tuning JDK for resource constrained containers

Alpaquita Linux
Revision 1.0
January 2024

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Contents

1. Overview

Worldwide trend to stay with application/services in containerized environment is not going to slow and will continue to drive upcoming software and IT infrastructure updates.

When speaking about containers people always think about microservices, microcontainers, or minimized resource usage. It is all true to some extent and indeed helps to manage resources more wisely using small containers tuned for particular application and cgroups functionality for controlling resources utilized by running containers. Control groups is a Linux kernel feature that helps to organize processes into hierarchical groups with usage of various resources that can be limited and monitored. It is supported now in a lot of Linux distributions including newer ones like Alpaquita. Since Alpaquita Linux provides container management engines, users can choose either one and enable cgroups to manage resources on a container host machine.

2. Control groups overview

Control groups v1 and v2 continue to exist in parallel and both are supported. This article provides examples based on Control groups v1 as the most mature technology that works well with Podman and Docker. Control groups v1 implementation provides a resource specific controller hierarchy. In other words, each resource, such as CPU, memory, I/O, and so on, has its own control group hierarchy.

Let's set up cgroups in Alpaquita Linux as an example. You can download Alpaquita Linux from the BellSoft website. It is a lightweight, small, and secure distribution tuned to run Java workloads. Any other Linux distribution is also suitable as long as it provides cgroups support and Podman/Docker packages.

```
sudo rc-update add cgroups
sudo rc-service cgroups start
```

The cgroups v1 can be set by editing `/etc/rc.conf`, assigning the `legacy` option to `rc_cgroup_mode`. If you choose to set up cgroups v2, you should assign `unified` option to `rc_cgroup_mode`. You can enable the controllers for cgroups v2 in the `rc_cgroup_controllers` parameter with the following options: `cpuset cpu io memory hugetlb pids`.

Example of `rc.conf` file modification:

```
# This sets the mode used to mount cgroups.
# "hybrid" mounts cgroups version 2 on /sys/fs/cgroup/unified and
# cgroups version 1 on /sys/fs/cgroup.
# "legacy" mounts cgroups version 1 on /sys/fs/cgroup
# "unified" mounts cgroups version 2 on /sys/fs/cgroup
rc_cgroup_mode="legacy"
```

You might need to run the following command for the changes to take effect or reboot the host:

```
sudo rc-service cgroups restart
```

For proper resource management with cgroups, use the corresponding OCI runtime in the setup. We recommend setting up Podman's `crun` runtime as the default runtime since it works perfectly with cgroups v1. Edit `/etc/containers/containers.conf` file and change `Default OCI runtime` value to be always `crun`.

Typical usage of resource management is to provide corresponding options in the `podman run` command. Few suitable options are listed in the table below that you can use right away in your experiments and setups.

Option	Description
–**cgroup-conf**=KEY=VALUE	When running on cgroups v2, specify the cgroup file to write to and its value. For example –cgroup-conf=memory.high=1073741824 sets the `memory.high` limit to 1GB.
–**cpu-period**=limit	Set the CPU period for the Completely Fair Scheduler (CFS), which is a duration in microseconds. Once the container's CPU quota is completely used, it will not be scheduled to run until the current period ends. Defaults value is 100000 microseconds.
–**cpu-quota**=limit	Limit the CPU Completely Fair Scheduler (CFS) quota.Limit the container's CPU usage. By default, containers run with the full CPU resource. The limit is a number in microseconds. If a number is provided, the container will be allowed to use that much CPU time until the CPU period ends (controllable via –cpu-period).
–**cpus**=number	Number of CPUs. The default is 0.0 which means no limit. This is shorthand for –cpu-period and –cpu-quota, therefore the option cannot be specified with –cpu-period or –cpu-quota.
–**cpuset-cpus**=number	CPUs in which to allow execution. Can be specified as a comma-separated list (e.g. 0,1), as a range (e.g. 0-3), or any combination thereof (e.g. 0-3,7,11-15).
–**memory**, **-m**=number[unit]	Memory limit. A unit can be b (bytes), k (kibibytes), m (mebibytes), or g (gibibytes).Allows the memory available to a container to be constrained.

For more information, see `podman run` documentation.

The following is an example of resource assignment options for a running container.

Typical container hosts have a lot of memory and CPU power, however, it is quite common to set limits for running containers, so all containers get enough resources they need. Here is an example of Podman container execution with cpu=1 and mem=1G limits.

```
podman run --cpus=1 --memory=1G -it --rm docker.io/bellsoft/liberica-runtime-
container:jdk-all-17-glibc catm/sys/fs/cgroup/memory/memory.limit_in_bytes
1073741824
podman run --cpus=1 --memory=1G -it --rm docker.io/bellsoft/liberica-runtime-
container:jdk-all-17-glibc cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
100000
```

But if you want to increase the CPU limit, update the option value accordingly.

```
podman run --cpus=2 --memory=1G -it --rm docker.io/bellsoft/liberica-runtime-
container:jdk-all-17-glibc cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
200000
```

3. JDK performance

When it comes to JDK we should consider some specifics to get the best of containerized applications or services. Even though it is all about microservices, microcontainers, and minimal resource usage, Java containers would suffer from overly limited resources provided to a running container. Let's explore one example related to Java behavior under reduced CPU and memory budget. The following example is about garbage collection that influences overall performance results.

Java version

Modern JDKs are designed to be container environment aware and friendly. Certain JDK versions are able to detect enforced resource quotas with Linux control group (cgroup) support. At the moment all current JDK versions, such as Liberica JDK 17, Liberica JDK 11.0.16+, and Liberica JDK 8u372+ as well as newer JDK versions support both cgroups v1 and cgroups v2 configurations. This support allows to detect that certain resource quotas are set up when running in a container, so that those quotas can be used for Java operations. Resource limits affect the garbage collector type activated by a JVM, the sizes of thread pools, the default size of the heap, and so forth.

Even though JDK knows that it is running in a container, some default options are not suitable for Java applications in a container. This is why it makes sense to additionally tune those options, namely `-XX:InitialRAMPercentage`, `-XX:MaxRAMPercentage`, and `-XX:MinRAMPercentage`.

These options are specified in percent, which is more preferable than setting maximum and minimum heap size for an application via `-Xmx` and `-Xms` options respectively. The `-XX RAMPercentage` options can set the heap size relative to the memory allocated to a container with the limit parameters and be updated automatically on redeployment.

In other words starting a container with either of the following commands makes it unnecessary to redefine `-Xmx` and `-Xms` in Dockerfile if they were supplied in a container. The `-XX:MaxRAMPercentage`, and `-XX:MinRAMPercentage` parameters set up in a Dockerfile stay unchanged.

```
podman run --memory=1G -it --rm docker.io/bellsoft/liberica-runtime-
container:jdk-all-17-glibc java -
XX:MaxRAMPercentage=50 -XX:MinRAMPercentage=50 -XX:+PrintFlagsFinal -version |
grep MaxHeapSize
   size_t MaxHeapSize                              = 536870912
{product} {ergonomic}
   size_t SoftMaxHeapSize                          = 536870912
{manageable} {ergonomic}
```

```
podman run --memory=2G -it --rm docker.io/bellsoft/liberica-runtime-
container:jdk-all-17-glibc java -
XX:MaxRAMPercentage=50 -XX:MinRAMPercentage=50 -XX:+PrintFlagsFinal -version |
grep MaxHeapSize
   size_t MaxHeapSize                              = 1067450368
{product} {ergonomic}
   size_t SoftMaxHeapSize                          = 1067450368
{manageable} {ergonomic}
```

Garbage Collector behavior

Garbage collection is an important and inevitable part of JVM that has effect on the overall performance of an application. It is very useful to know which Garbage Collector (GC) is used by the JVM running in a container. To check the type of GC, use the `-Xlog:gc=info` command. For example, when container limits the use to a single CPU, the Serial GC is selected. If more than one CPU is active and sufficient memory (at least 2GB) is allocated to the container, the G1 GC is selected in a container friendly Java version, such as version 11 or later. Note the selected GC depending on the CPU settings in the following examples:

```
podman run --cpus=1 --memory=2G -it --rm docker.io/bellsoft/liberica-runtime-
container:jdk-all-17-glibc java -Xlog:gc=info -version
[0.003s][info][gc] Using Serial
openjdk version "17.0.6" 2023-01-17 LTS
OpenJDK Runtime Environment (build 17.0.6+10-LTS)
OpenJDK 64-Bit Server VM (build 17.0.6+10-LTS, mixed mode)
```

```
podman run --cpus=2 --memory=2G -it --rm docker.io/bellsoft/liberica-runtime-
container:jdk-all-17-glibc java -Xlog:gc=info -version
[0.003s][info][gc] Using G1
openjdk version "17.0.6" 2023-01-17 LTS
OpenJDK Runtime Environment (build 17.0.6+10-LTS)
OpenJDK 64-Bit Server VM (build 17.0.6+10-LTS, mixed mode)
```

Serial GC is the oldest garbage collection mechanism existing from the early days of Java. It can be suitable for memory and CPU constraint devices, but there will be long pauses in application work especially if significant amount of memory is involved. If you want your application or service to have lower response latency and to run longer without interruption, assign more resources to take advantage of better performing GC types. JDK can enable G1 GC automatically if resources reach certain limits. G1 garbage collector has more predictable pause time while achieving higher throughput.

Performance impact

Running G1 garbage collector provides performance benefits over Serial GC. Benchmark results can help to see how it all works. The single-threaded benchmark is suitable to run in both configurations with `--cpus=1` and `--cpus=2` as long as the memory limit is the same. For simplicity, we chose the JMH project and corresponding sample benchmarks. The container with JDK 17 (Liberica JDK build 17.0.6+10-LTS) was used to set up JMH and pre-built benchmarks in the default location. You can build the `jmh-samples` benchmarks from the Java Microbenchmark Harness (JMH) project.

Clone the repository and build the existing sample benchmarks as follows:

```
git clone https://github.com/openjdk/jmh
cd jhm
mvn clean verify
```

The container with JHM benchmarks was created, and we can run it multiple times to collect necessary data for comparison choosing one benchmark for this purpose since the complete set would execute for too long.

Since we want to see the difference in the single-threaded benchmark with different garbage collector types, 2GB of memory was specified as the least possible for G1 to be activated. Selecting either 1 CPU or 2 CPUs activates Serial GC or G1 respectively. Specify 2 CPUs to activate G1 Garbage Collector.

```
podman run --cpus=2 --memory=2G -it --rm 5d412591e12c java -jar
/root/jmh/test/target/benchmarks.jar
org.openjdk.jmh.samples.JMHSample_25_API_GA
```

Specify 1 CPU with the same memory for the JVM to use Serial GC.

```
podman run --cpus=1 --memory=2G -it --rm 5d412591e12c java -jar
/root/jmh/test/target/benchmarks.jar
org.openjdk.jmh.samples.JMHSample_25_API_GA
```

CPU option / GC type	Benchmark results
–cpus=1 –memory=2G, SerialGC	646749.890 ops/s
–cpus=2 –memory=2G, G1GC	694589.369 ops/s

The benchmark results show that G1 performs better than Serial GC. Assigning 2 CPUs even for a single-threaded applications or services can result in better performance. Overall performance difference is about 4-7% measured multiple times under different conditions. The absolute numbers can vary slightly depending on the system condition and CPU governor setup at the time.

Native Image

Liberica Native Image Kit (NIK) is based on GraalVM CE Open Source project and delivers a utility capable of converting your JVM-based application into a fully compiled native executable ahead-of-time under the closed-world assumption with an almost instant startup time. GraalVM is a high-performance JDK designed to accelerate Java application performance while consuming fewer resources. GraalVM offers two ways to run Java applications: on the HotSpot JVM with Graal just-in-time (JIT) compiler or as an ahead-of-time (AOT) compiled native executable. Liberica Native Image Kit (NIK) is delivered as a package that could be installed on your build host since it will not be needed at runtime once an application is built as a native image.

The GraalVM Community Edition (the current latest 22.3.2 version) supports Serial GC and Epsilon GC that performs slower even if more resources are assigned to a container with an application or service. The upcoming GraalVM 23.x releases will include "parallel" GC implementation that can be enabled using the `--gc=parallel at build time` option at the image build time, and the number of worker threads can be set at runtime using the `-XX:ParallelGCWorkers` option. Worker threads are started early during application startup.

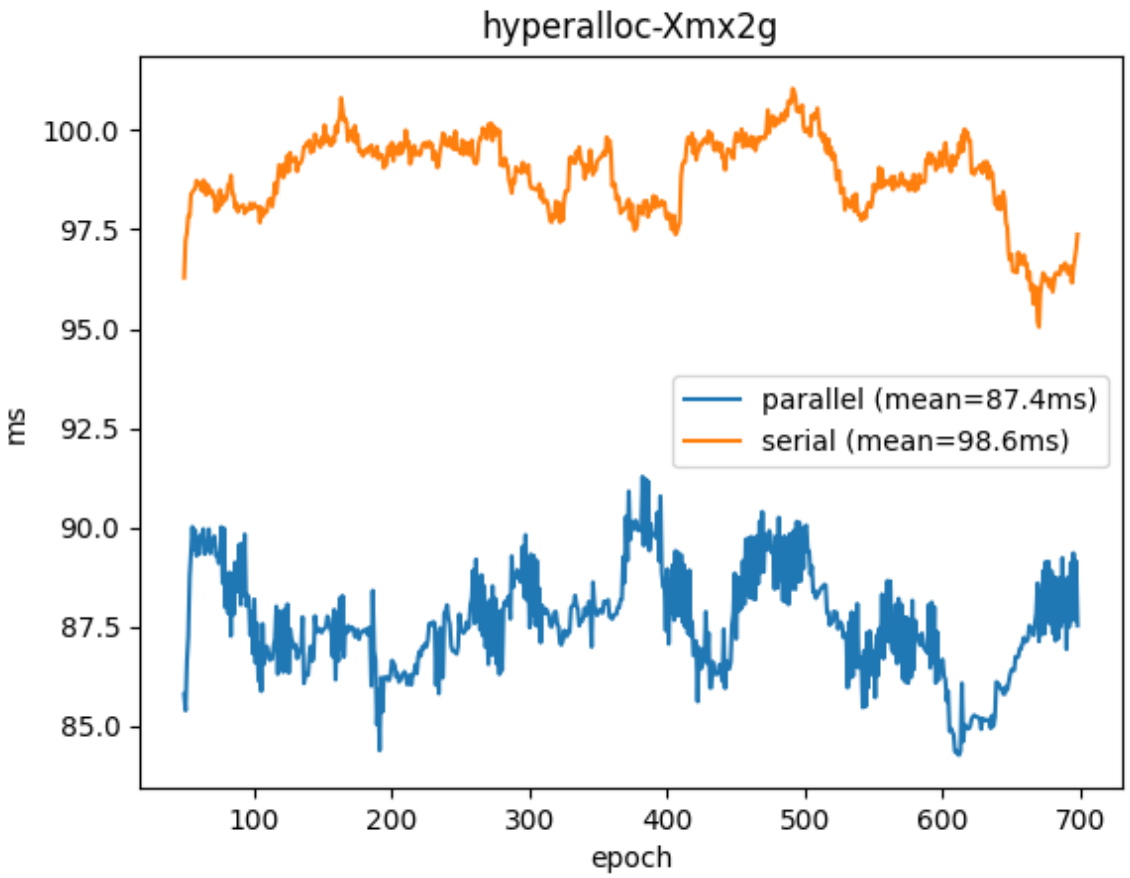
For more information about Parallel GC for GraalVM, see Parallel garbage collector.

Performance impact

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

The new "parallel" GC implementation extends a variety of GC types available in GraalVM Community Edition 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.



The picture and calculated mean values show that "parallel" GC implementation provides less pause time, thus improving performance and latency times for application/service running in similar configurations with two or more available threads or CPU.

4. Conclusion

Containerized applications and services do not have to be always running with minimal resources assigned to a container unless it is strictly required. In many cases, Java applications or services perform better and have better response time if slightly more resources are allocated for a running container, so that a more suitable GC type is eventually used. We recommend having at least two CPU and 2GB memory limits set for a running container with Java workloads. In some cases, you might want to choose Native Image Kit to get faster startup and lower memory consumption at runtime, but this is a topic for another article.

Alpaquita Linux
Tuning JDK for resource constrained containers

bellsoft