Java Garbage Collection Characteristics and Tuning Guidelines for Apache Hadoop TeraSort Workload

Shrinivas Joshi, Software Performance Engineer
Vasileios Liaskovitis, Performance Engineer

1. Introduction

In this article, we take a detailed look at the garbage collection (GC) characteristics of TeraSort[1] workload running on top of an Apache Hadoop[2] framework deployed on a seven-node cluster. Apache Hadoop is a Java™ technology-based framework that facilitates distributed computing using commodity hardware. TeraSort workload is an example MapReduce application that ships with the Apache Hadoop distribution and is intended for sorting terabytes of data. We will discuss our recommendations for Java Virtual Machine (JVM) flags that can help tune GC behavior on a similar TeraSort setup. We will show how GC tuning helped us achieve as much as a 7% gain in TeraSort's performance on our experimental cluster. We will also discuss GC characteristics and associated tuning guidelines for both *Map* and *Reduce* task JVMs.

Included is a primer of associated technologies and terms referred to in this article. Readers who are well-versed in the areas of distributed computing using Apache Hadoop and Java GC can skip relevant sections.

The rest of the article is organized as follows. Section 2 provides a brief introduction to the Apache Hadoop framework and MapReduce technology. Section 3 describes the experimental setup. Section 4 talks about GC characteristics and tuning guidelines for *Map* and *Reduce* JVM processes. Section 5 analyzes the performance impact of GC tuning. Section 6 summarizes the findings and discusses next steps. Appendix A provides a glossary of technical terms associated with Java garbage collection. We advise the reader to review Appendix A before reading Section 4 if he/she is not familiar with Java GC terms.

2. Apache Hadoop and MapReduce Technology

Apache Hadoop is a distributed computing framework designed to run highly parallel workloads on clusters of commodity hardware systems. Hadoop supports applications developed using a programming model called MapReduce, briefly explained later in this section. Using this model, the user

[1] http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html
[2] http://hadoop.apache.org/

implements a job/application by providing what are called *Map* functions and *Reduce* functions. The Hadoop framework then divides these jobs into a number of smaller, independent tasks that can be executed in parallel on different nodes of the cluster. The Hadoop framework transparently provides the ability to schedule, execute, and re-execute (in case of failures) these tasks. Hadoop also provides a distributed file system called Hadoop Distributed File System (HDFS) to facilitate data storage across the cluster to increase data locality and improve fault tolerance. This framework is implemented using the Java™ programming language, and the *Map* and *Reduce* tasks are executed on top of JVMs.

As noted previously, MapReduce is a programming model that facilitates parallel processing of large amounts of data in a clustered environment. In the MapReduce model, there are two different types of activities: *Map* and *Reduce*.

Map, written by the user, takes an input key/value pair and produces a set of *intermediate* key/value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the *Reduce* function. The *Reduce* function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per *Reduce* invocation.[3] Note that some usage scenarios of the MapReduce programming model could yield more than one output value from *Reduce* function.

3. Experimental Setup

Before diving into the details, let us first look at the hardware and software configuration of the cluster under test (CUT) used for this study, and the relevant Hadoop framework configuration settings.

3.1 Cluster Under Test

This is the hardware and software configuration of the CUT and its associated components:

- 7 nodes: 1 name node and 6 data nodes.
- 2 chips/6 cores per chip: AMD Opteron™ 2435 @ 2600 MHz.
- 5 SATA disks @ 7200 RPM per node: 3 disks exclusively used for HDFS storage, 1 disk exclusively used for logging and launching of Hadoop framework, and 1 disk used for OS as well as HDFS storage (OS and HDFS sit on separate partitions of the same disk).
- 16 GB RAM.
- SUSE Linux Enterprise Server 10 (x86_64) SP2, Kernel 2.6.16.46-0.12-smp.
- Java version "1.6.0_16"
  Java(TM) SE Runtime Environment (build 1.6.0_16-b01)
  Java HotSpot(TM) 64-Bit Server VM (build 14.2-b01, mixed mode)
- Hadoop version 0.20.3-dev.
- Input data set size used by TeraSort: 64 GB.

[3] http://labs.google.com/papers/mapreduce-osdi04.pdf

3.2 Hadoop Configuration

This section describes some values of the Hadoop framework properties relevant to this article. Please note that the descriptions of these properties are as noted in the appropriate Hadoop configuration files.

- ***mapred.job.reuse.jvm.num.tasks***: How many tasks to run per JVM. If set to -1, there is no limit. We set this to -1.
- ***mapred.tasktracker.map.tasks.maximum***: The maximum number of map tasks that will be run simultaneously by a task tracker. In our case, we set this to 12.
- ***mapred.tasktracker.reduce.tasks.maximum***: The maximum number of reduce tasks that will be run simultaneously by a task tracker. In our case, we set this to four.
- ***io.sort.mb***: The total amount of buffer memory to use while sorting files, in megabytes. By default, gives each merge stream 1MB, which should minimize seeks. In our case, we set this to 420 MB.
- ***io.sort.factor***: The number of streams to merge at once while sorting files. This determines the number of open file handles. In our case, we set this to 100.
- ***dfs.block.size***: The default block-size for new files. We set this to 268,435,456 (256 MB) in our case.

3.3 Baseline Configuration

To get the base understanding of GC behavior, we set the maximum heap size of *Map* and *Reduce* JVMs to a value that would be acceptable based on the total size of memory we wanted to make available to all the JVM processes. This turned out to be 700 MB in our case. So we were using 12 * 700M for *Map* JVMs and 4 * 700M for *Reduce* JVMs. Thus, out of the total 16 GB available on each of the nodes, about 11 GB was used for Java processes. Given the I/O-intensive nature of TeraSort, we wanted to leave the remaining 5 GB of memory for disk cache and other OS processes. We also did not set initial heap size since we wanted to allow the JVM to grow the heap requirements optimally over a period of time. That way, any possible free memory could be used by other processes.

Based on GC ergonomics, parallel GC with 10 parallel collector threads was the choice made by the JVM for its GC flags. Thus our baseline JVM flags were "`-server -Xmx700M -XX:+UseParallelGC -XX:ParallelGCThreads=10`". The GC logs we studied here were generated using "-verbose:gc -Xloggc:<path_to_GC_log_folder>/@taskid@.gc -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps" JVM flags.

4. GC Characteristics and Tuning Suggestions

In this section we describe the GC characteristics of *Map* and *Reduce* JVM processes we observed through analysis of JVM-generated GC logs. We also talk about GC tuning guidelines we came up with based on the findings of this analysis.

4.1 GC Characteristics of *Map* JVMs

Here is what we observed with GC behavior of *Map* JVMs by looking at GC logs:

- Within the first few seconds of the *Map* JVM run, the old generation occupancy goes up to the range of approximately 420M to 440M. Also, this huge chunk of data gets allocated directly into the old generation; perhaps a huge array or a collection object backed by an array gets allocated that cannot fit into the young generation.
- No collections happen for approximately the next 20 to 40 seconds. After this, a major collection collects almost everything from the old generation. Somewhere between 2M and 22M data is left live in the old generation after this major collection.
- Old generation occupancy stays in this low-20M range for about 20 to 80 seconds longer. After this, another huge allocation happens in the old generation. Old generation occupancy goes back up to the 420M-to-440M range.
- Old generation occupancy stays in this 420-440M range for approximately 30 to 120 seconds longer. After this, another major collection happens, and old generation occupancy returns to the 2-20M range.
- This pattern repeats throughout the run of *Map* JVMs. We think this repetitive GC pattern coincides with multiple *Map* tasks being scheduled on the same JVM throughout the execution.

Fig. 1 contains a graph generated using the GCViewer[4] tool that shows the observed GC behavior for one of the indicative *Map* JVMs running on one of the nodes.

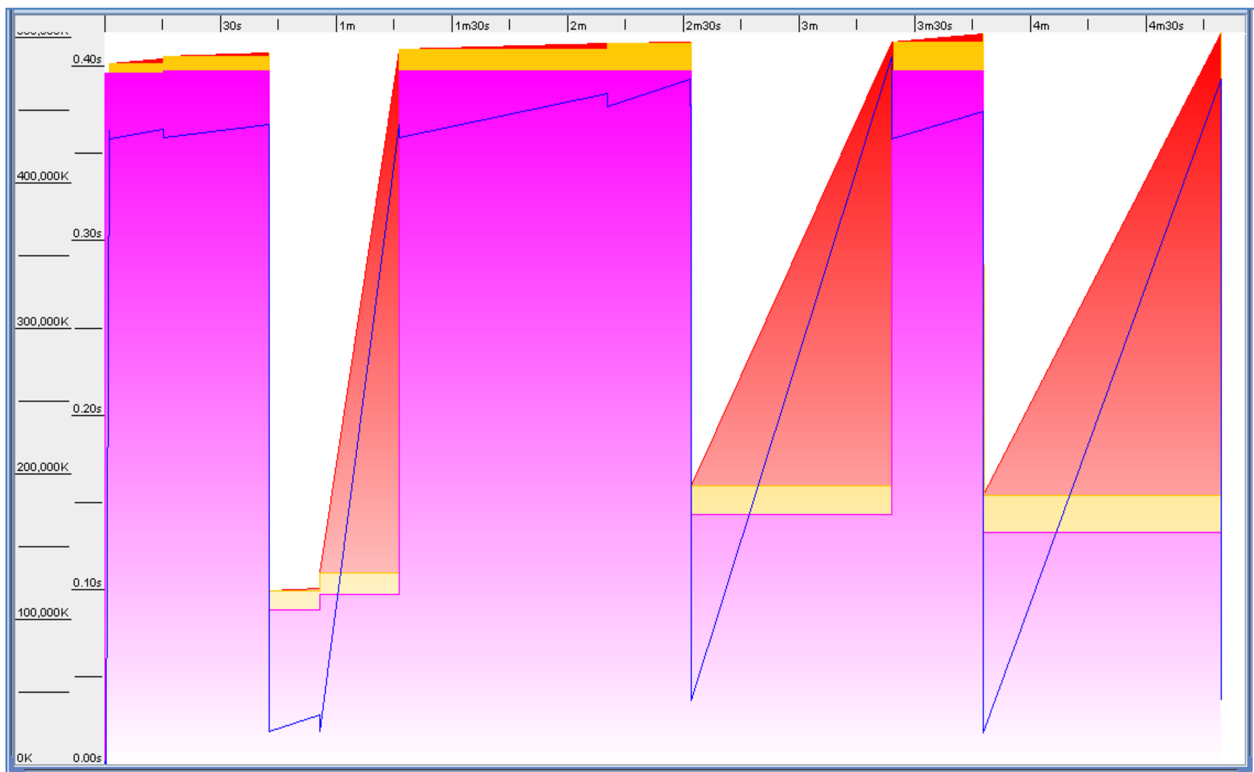

Figure 1: Occupancy of different generations of Java heap over the period of an indicative *Map* JVM run.

[4] http://www.tagtraum.com/gcviewer.html

The legend used by GCViewer tool for Fig. 1:

Total size of old generation
Total size of young generation
Size of total heap
Used heap

Fig. 2 shows the timeline of minor and major collections and corresponding pause times for the same *Map* JVM run shown in Fig. 1. This graph was generated using the GCHisto tool.[5]

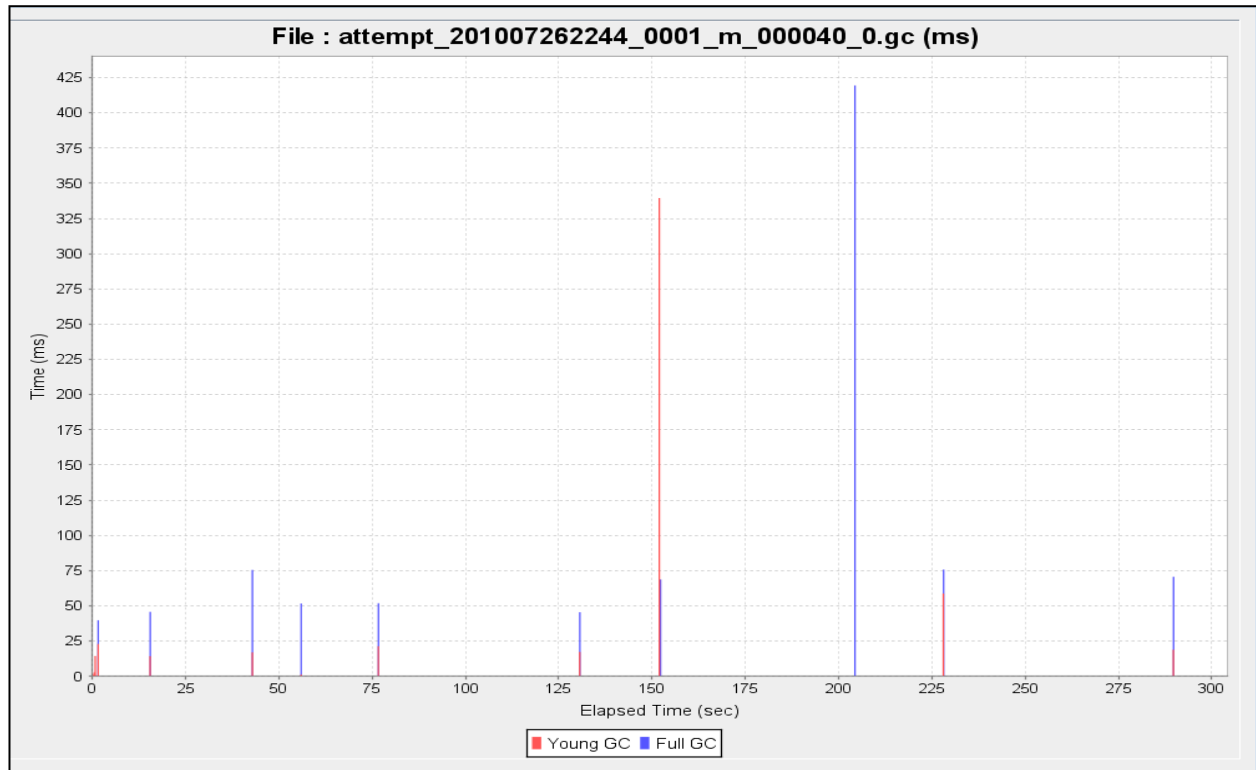

Figure 2: GC timeline of an indicative *Map* JVM run.

4.2 GC Tuning Guidelines for *Map* JVMs

Based on these and other observations described in this section, we developed a set of guidelines on tuning GC behavior of *Map* JVMs on a similar TeraSort setup:

- ***Use parallel generational collector with four parallel GC threads***: In analyzing GC logs of all *Map* JVMs, we noticed 606 major collections and 895 minor collections. It was clear the *Map* JVMs were not GC-heavy, so we decided to bump down the number of parallel GC threads to four instead of the default 10 threads. The intention was to reduce resource contention and fragmentation resulting from multiple parallel GC threads. Since we were not concerned about latency or pause times, we decided to continue using the parallel generational collector.

[5] https://gchisto.dev.java.net/

- ***Tune initial and max heap size and disable adaptive sizing***: Some of the minor collections were due to smaller young generation, which resulted from the adaptive sizing policy used by the JVM. To avoid these redundant minor collections, we decided to disable adaptive sizing of the heap generations and to set the initial heap size equal to maximum heap size.
- ***Disable explicit GC***: To avoid interference from programmatically triggered GC, we decided to disable explicit GC.
- ***Tune initial and maximum young generation size***: A maximum of about 30M data was noted live in young generation at any given point in time. Thus, we decided to set initial and maximum young generation sizes to 60M, which gave us sufficiently large young generation to prevent triggering unnecessary minor collections.
- ***Tune permanent generation size***: Based on the GC ergonomics policy used by the JVM, 21M was reserved for permanent generation, but no more than 10M of data was allocated into permanent generation. Thus, we decided to set initial and maximum permanent generation size to 12M.
- ***Tune survivor space size***: Not more than 2M of data survived after minor collections, so we decided to set the initial survivor space size to something slightly larger than 2M. We did this by setting the initial survivor ratio to 28, which made the *from* and *to* survivor spaces end up being 2,176K.

Based on these observations, here is the complete set of tuned GC and heap flags for *Map* JVMs:

```
-server -Xms700M -Xmx700M -XX:+UseParallelGC -XX:ParallelGCThreads=4 -
XX:-UseAdaptiveSizePolicy -XX:+DisableExplicitGC -XX:NewSize=60M -
XX:MaxNewSize=60M -XX:PermSize=12M -XX:MaxPermSize=12M -
XX:InitialSurvivorRatio=28
```

4.3 GC Characteristics of *Reduce* JVMs

Here is what we observed with GC behavior of *Reduce* JVMs by looking at GC logs:

- Within the first 8 to 15 seconds of the run, the old generation occupancy reaches up to 440M. At this point, there is between 30 and 40 seconds high allocation (up to 220M/500 ms) of very short-lived objects that keep getting reclaimed completely. During this time, the old generation occupancy stays constant. This results in a number of full GC cycles.
- At around 45 seconds into the run, the old generation is collected and its occupancy drops to anywhere between approximately 134M and 200M.
- Between the next 25 seconds and 50 seconds, the old generation keeps filling up and returns to about 440M occupancy.
- After this, there is again between 30 and 40 seconds high allocation (up to 220M/500 ms) of very short-lived objects, which keep getting reclaimed completely. During this, the old generation occupancy stays constant at about 440M. Again, this results in a number of full GC cycles.
- This pattern keeps repeating throughout the run of the *Reduce* phase.

Fig. 3 contains a graph generated using the GCViewer tool that shows the observed GC behavior for one of the indicative *Reduce* JVMs running on one of the nodes. It uses the same legend as Fig. 1.

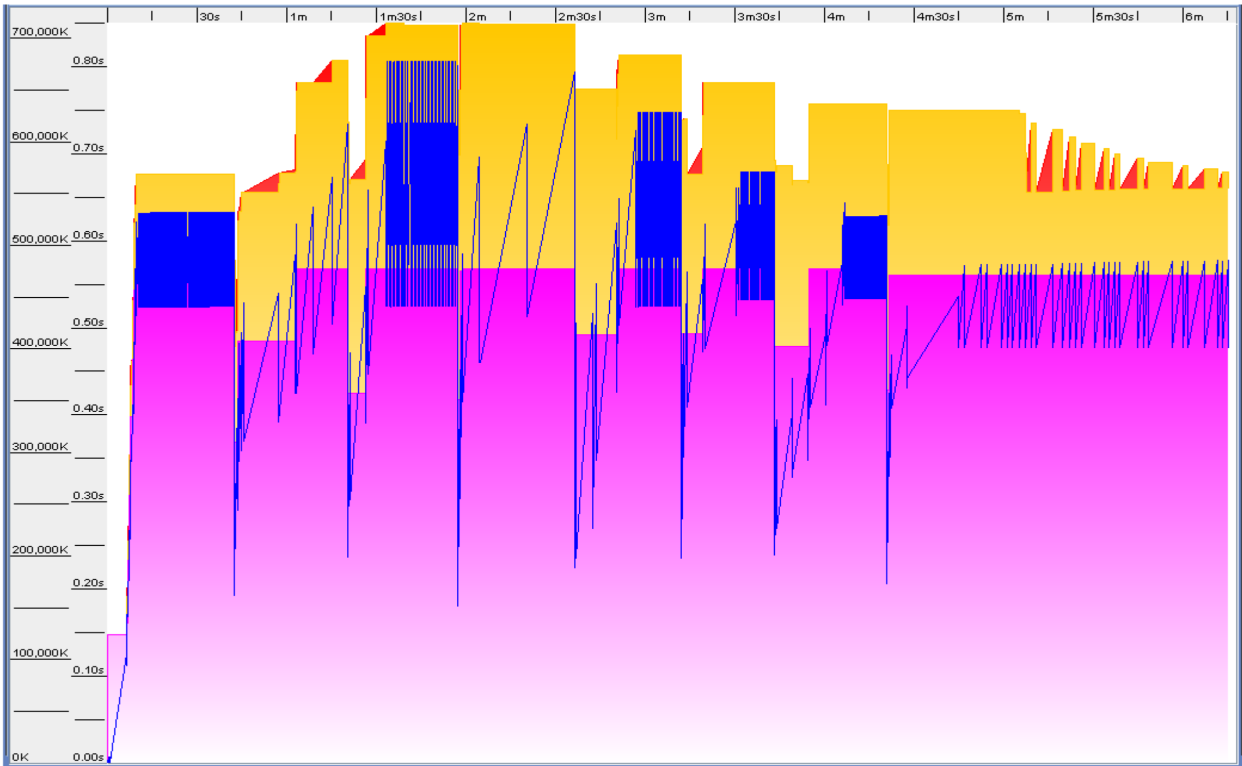

Figure 3: Occupancy of different generations of Java heap over the period of an indicative *Reduce* JVM run.

Fig. 4 shows the timeline of minor and major collections and corresponding pause times for the same *Reduce* JVM run shown in Fig. 3. This graph was generated using GCHisto.

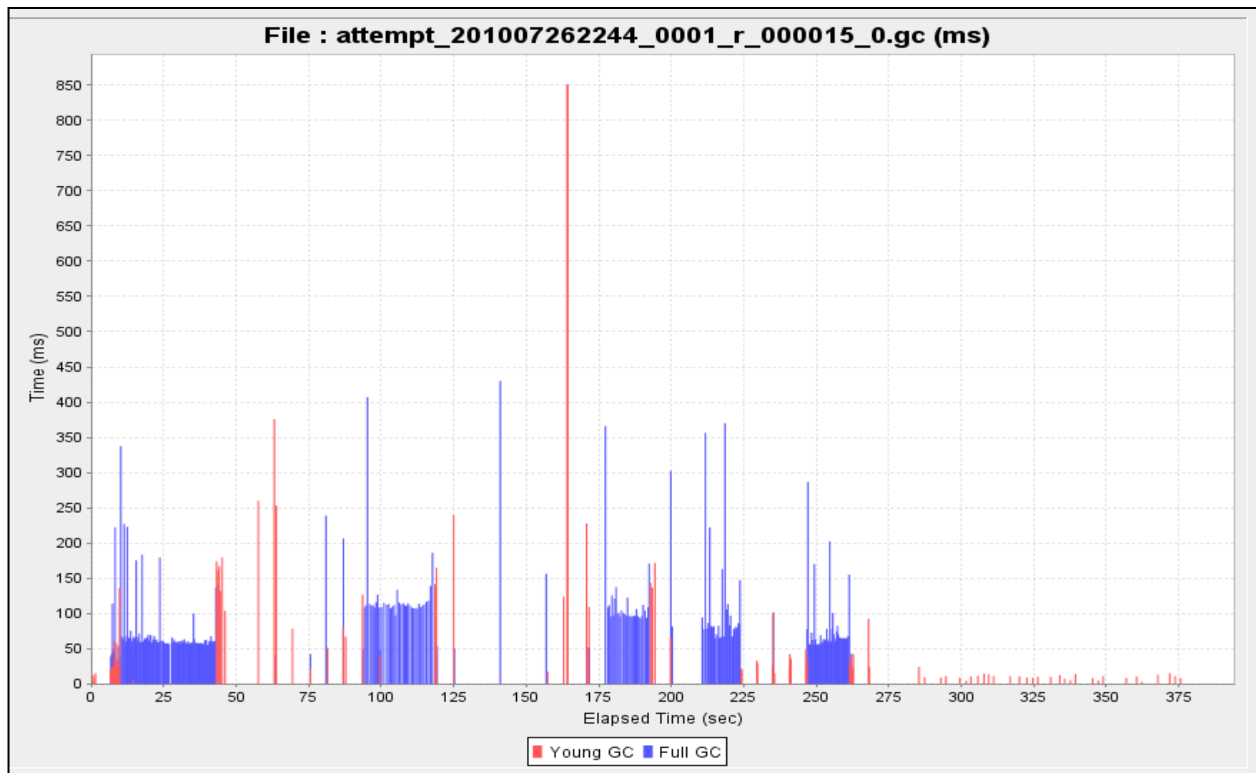

Figure 4: GC timeline of an indicative *Reduce* JVM run.

4.4 GC Tuning Guidelines for *Reduce* JVMs

Based on these and other observations described in this section, we developed a set of guidelines for tuning GC behavior of *Reduce* JVMs on a similar TeraSort setup:

- ***Use parallel generational collector with eight parallel GC threads***: GC activity in *Reduce* JVMs is much more prominent compared to GC activity in *Map* JVMs. A lot of adaptive sizing of young generation and survivor spaces happens during the *Reduce* phase. The *Reduce* phase showed a high allocation rate as well as a high survival rate. We also noticed that there were far too many major collections relative to minor collections. In analyzing the GC logs of all the *Reduce* JVMs, we noticed there were 11,252 major collections and 3,019 minor collections. As mentioned previously, we were not concerned about latency or pause times, so decided to continue using parallel generational collector. However, since as many as four *Reduce* JVMs could be active at one time, we decided to reduce the number of parallel GC threads slightly from the default value of 10; we set it to eight.
- ***Tune initial and max heap size, young generation size, and disable adaptive sizing***: We noticed a high young generation allocation rate that frequently also got reclaimed completely. As much as 220M worth of objects were getting allocated and reclaimed from the young generation every 500 ms during certain periods of the *Reduce* phase. Also, before reaching this allocation rate, the JVM was resizing the young generation quite often, which led to frequent minor collections. To avoid these collections and to avoid excessive young generation resizing, we decided to set initial heap size equal to maximum heap size, disable adaptive sizing, and set the initial and maximum young generation size to 240M. Although we think that increasing young generation beyond 240M would help more in this situation, we did not do so. We chose not to

increase young generation to more than 240M because we wanted the rest of the heap to be reserved for old generation, since we noticed a high survival rate and we did not want to increase the total heap size beyond 700M.
- ***Disable explicit GC***: To avoid interference from programmatically triggered GC, we decided to disable explicit GC.
- ***Tune permanent generation size***: Based on the GC ergonomics policy used by the JVM, 21M was reserved for permanent generation, but no more than 10M of data was allocated into permanent generation. Thus we decided to set initial and maximum permanent generation size to 12M.

Based on these observations, here is the complete set of tuned GC and heap flags for *Reduce* JVMs:

```
-server -Xms700M -Xmx700M -XX:+UseParallelGC -XX:ParallelGCThreads=8 -
XX:-UseAdaptiveSizePolicy -XX:+DisableExplicitGC -XX:NewSize=240M -
XX:MaxNewSize=240M -XX:PermSize=12M -XX:MaxPermSize=12M
```

Note that, even though there were a large number of major collections, we did not see much gain from using the parallel old generation collector (`-XX:+UseParalleOldGC`).

5. Impact of GC Tuning

As noted in previous sections, there were enough differences in the GC characteristics of *Map* and *Reduce* JVMs that they warranted a way of passing different sets of JVM flags to *Map* and *Reduce* JVMs. The version of the Hadoop framework we used did not support this feature. We made appropriate changes to the Hadoop source code to implement this feature and passed the previously mentioned tuned set of flags for *Map* and *Reduce* JVMs separately.

After using the tuned set of JVM flags, we noticed the following changes in GC characteristics of TeraSort workload:

Tuned* Map *JVMs:

- 264 major collections, compared to 606 major collections noted with the baseline configuration.
- 851 minor collections, compared to 895 minor collections noted with the baseline configuration.
- Total GC pause time (as calculated by aggregating *real* pause times reported by GC logs) during a particular run of the TeraSort workload was 103 seconds, compared to 138 seconds with the baseline configuration.

Tuned* Reduce *JVMs:

- 6,309 major collections, compared to 11,252 major collections noted with the baseline configuration.
- 1,478 minor collections, compared to 3,019 minor collections noted with the baseline configuration.

- Total GC pause time (as calculated by aggregating *real* pause times reported by GC logs) during a particular run of the TeraSort workload was 836 seconds, compared to 1,173 seconds with the baseline configuration.

Table 1 summarizes these differences.

Table 1: Performance impact of GC tuning

Phase	**Major Collections**		**Minor Collections**		**Total Pause Time**	
	Baseline	**GC-Tuned**	**Baseline**	**GC-Tuned**	**Baseline**	**GC-Tuned**
Map	606	264	895	851	138 sec	103 sec
Reduce	11,252	6,309	3,019	1,478	1,173 sec	836 sec

We can visualize these differences in GC behavior by comparing baseline and GC-tuned graphs for occupancy and timeline.

Fig. 5 shows how heap generation occupancy compares between baseline and GC-tuned runs of indicative *Map* JVM runs. The top part of the figure refers to the baseline configuration and the bottom part refers to the GC-tuned configuration.

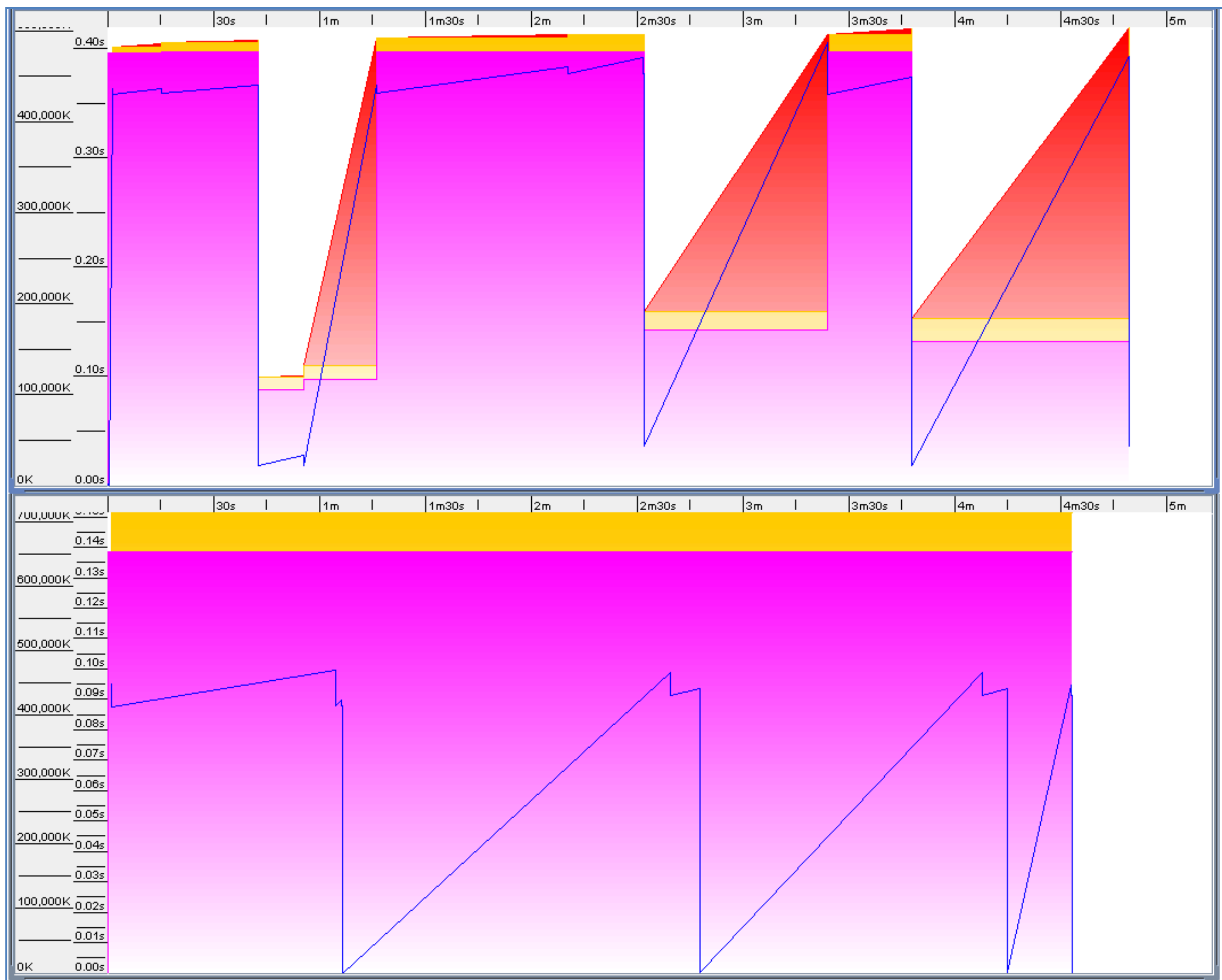

Figure 5: Comparison of Java heap generation occupancy of an indicative baseline *Map* run and an indicative GC-tuned *Map* run.

Fig. 6 shows timeline and corresponding pause times for minor and major collections occurring during the course of executing indicative *Map* JVM runs. Again, the top part of the figure refers to the baseline configuration and the bottom part refers to the GC-tuned configuration.

Here you can see that the frequency of major collections has gone down substantially. However, one thing to note is that with GC tuning, the average pause times for minor collections has also increased. The result, however, is a reduction in total pause time due to the reduced number of total collections.

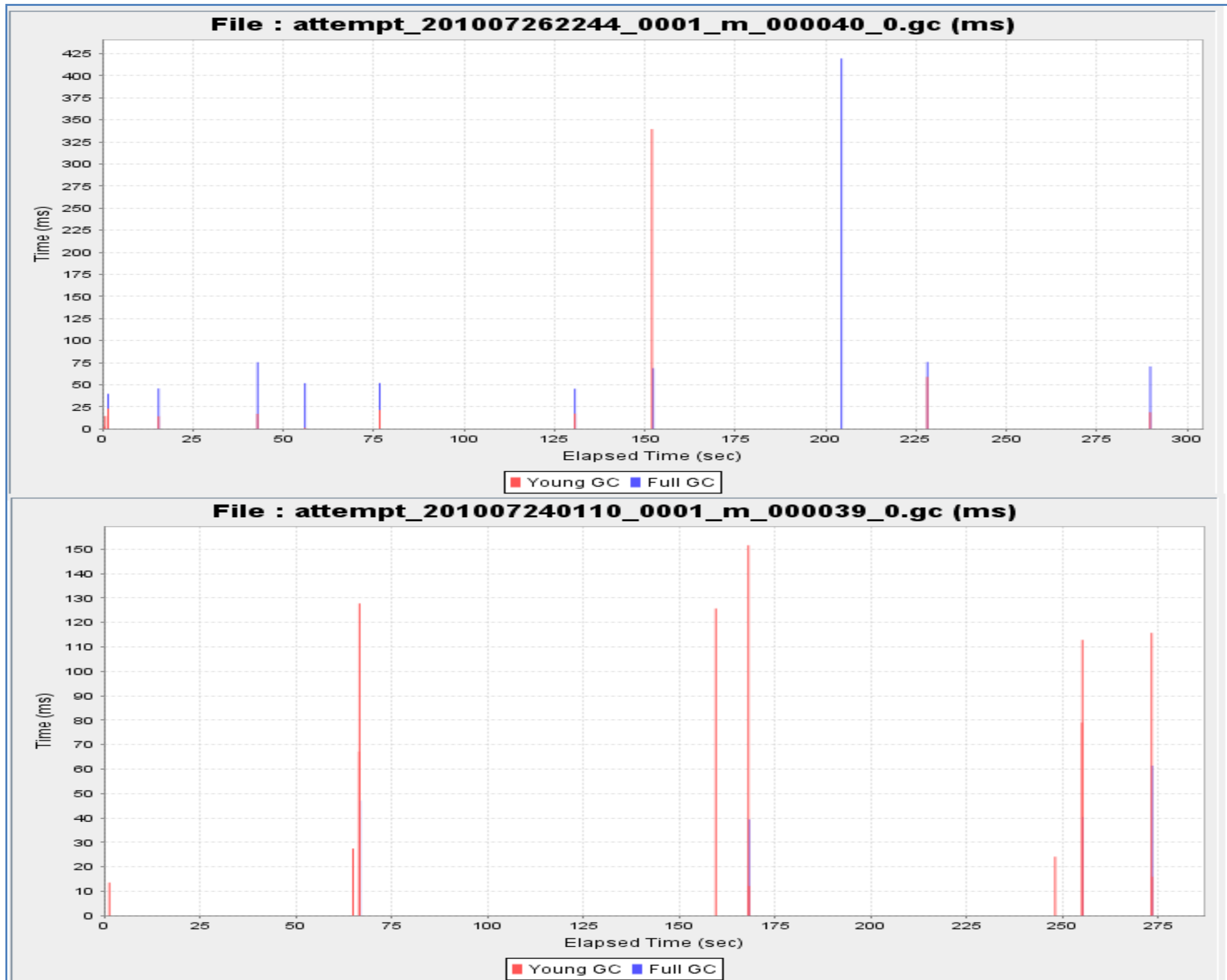

Figure 6: Comparison of GC timeline of an indicative baseline *Map* run and an indicative GC-tuned *Map* run.

Fig. 7 shows how heap generation occupancy compares between baseline and GC-tuned runs of indicative *Reduce* JVM runs. The top part of the figure refers to the baseline configuration and the bottom part refers to the GC-tuned configuration.

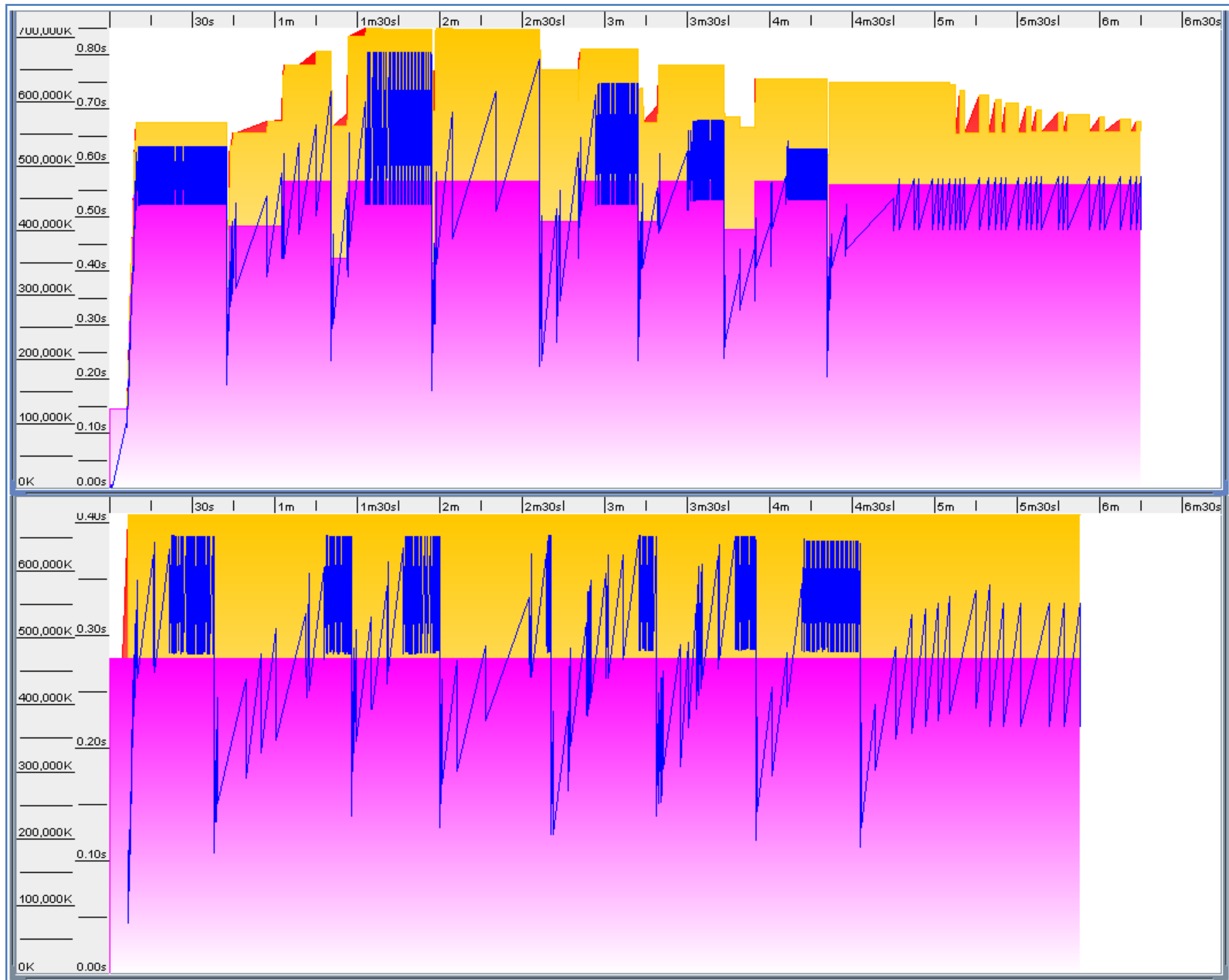

Figure 7: Comparison of Java heap generation occupancy of an indicative baseline *Reduce* run and an indicative GC-tuned *Reduce* run.

Fig. 8 shows timeline and corresponding pause times for minor and major collections occurring during the course of executing indicative *Reduce* JVM runs. Again, the top part of the figure refers to the baseline configuration and the bottom part refers to the GC-tuned configuration.

The frequency of minor and major collections has gone down substantially. However, as noted with *Map* JVMs, the average pause times for minor and major collections has also increased. The result, however, is a reduction in total pause time due to significantly fewer collections.

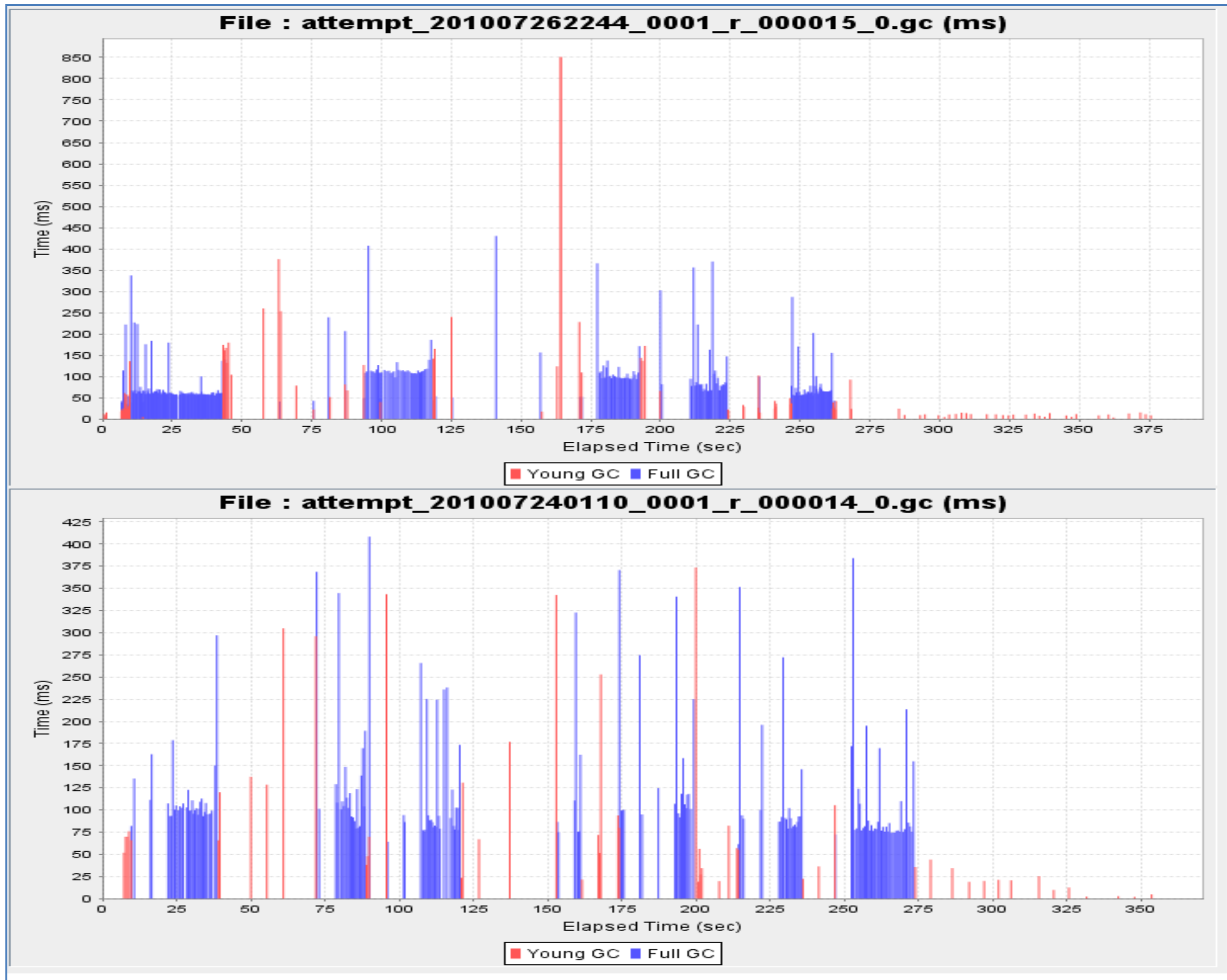

Figure 8: Comparison of GC timeline of an indicative baseline *Reduce* run and an indicative GC-tuned *Reduce* run.

We noticed some run-to-run variance in the TeraSort execution time. To understand the impact of GC tuning on TeraSort's execution time, we executed it 15 times and looked at best-case numbers and average-case numbers. With GC tuning, we saw a best-case gain of 7.45% and an average-case gain of 2.15% in terms of reduced total execution time.

Given the parallel nature of Hadoop workloads, and dependencies between different phases of Hadoop workload execution, it is difficult to achieve gains in execution time identical to the savings in GC pause times. In our case, looking at the pause times for both *Map* and *Reduce* JVMs, we saved 372 seconds worth of computing cycles (35 seconds with *Map* JVMs + 337 seconds with *Reduce* JVMs) across the cluster. That works out to 62 seconds per data node, because we had six of them. Note that without GC tuning, we noted an average execution time for TeraSort workload at around 767 seconds. If we scale these gains to a 1,000-node cluster, ignoring the performance effects that network bottlenecks could have in such a big cluster and assuming it is running a similar (although scaled up in terms of total input data set size) TeraSort workload that takes approximately 767 seconds to complete, then we are looking

at saving more than 17 hours of compute cycles[i] per execution. This is equivalent to shutting down 80 nodes in the cluster[ii].

We also measured power consumption on one of the AMD Opteron 2435 processor-based data nodes in the cluster while running TeraSort without GC tuning flags. Based on the power readings spanning a complete run, we noted the system drew 320.3W of power on average. Again, assuming a 1,000-node cluster and assuming the power consumption on all the data nodes is identical, we estimate reduced execution time from GC tuning would translate to a saving of as much as $22,996[iii] in annual operating expense. (Please see endnotes for more details on how we derived energy savings estimate.) Also, note that the hardware systems we used for these experiments are older-generation platforms, which may not be as energy-efficient as current-generation platforms. The energy savings estimates mentioned here are based on certain assumptions; your mileage may vary based on the hardware configuration and the execution characteristics of the workloads running on those hardware systems.

6. Conclusion and Next Steps

This article looked at how we analyzed JVM-generated GC logs to come up with a tuned set of JVM flags to improve GC behavior of a Hadoop TeraSort workload. It helped achieve as much as a 7.45% gain in total execution time. We also were able to save 62 seconds worth of computing cycles per data node, which translates to savings of more than 17 hours of computing cycles on a 1,000-node cluster based on similar hardware configurations. Small per-node gains easily translate into huge savings in terms of compute cycles and operating cost when we consider large-scale clusters, which are becoming more and more common.

As for next steps, we plan to move to the latest GA release of Oracle (HotSpot VM-based) JDK, which has been proven stable for Hadoop environments. We want to study if it is any different in terms of GC performance. We also want to understand how much more an increase in total heap size for *Reduce* JVMs might help in TeraSort performance. To make optimum gains from increased heap sizes for *Reduce* JVMs, we might have to make appropriate changes to other Hadoop properties as well.

Another GC optimization that is evident from the occupancy graphs is that the max heap requirements for *Map* JVMs could be reduced further. As shown in Fig. 1, no more than about 500M of total heap is ever used by *Map* JVMs.

We are also interested in exploring the impact of Hadoop configuration parameter values on garbage collection characteristics and how it affects our GC tuning guidelines. Apart from GC tuning, we are also interested in looking at other JVM- and Hadoop-level tuning opportunities, as well as OS- and hardware-level tuning opportunities. We will certainly keep you posted on our new findings.

We hope this article gives you some insight into TeraSort GC behavior and helps you tune it a little better. We will be glad to hear your comments/experiences in this area.

Appendix A. Glossary of Garbage Collection Terms

This section covers some of the technical terms associated with Java garbage collection referred to in this article.

- ***Adaptive Sizing***: Dynamic growing and shrinking of different generations and survivor spaces performed by JVMs based on GC behavior of the application.
- ***Collector***: Garbage collection algorithm.
- ***Explicit GC***: Programmatically triggered GC using calls to *System.gc()* method.
- ***Full/Major Collections***: Garbage collections that purge unused objects from the entire heap including young, old, and permanent generations.
- ***Generational Collector***: GC algorithms that divide Java heap into multiple chunks for GC activities. These heap chunks are referred to as generations.
- ***Minor Collections***: Garbage collections that purge unused objects from only the young generation.
- ***Old Generation***: Java heap generation where long-lived objects reside. Sometimes huge arrays that cannot fit in young generations are allocated directly into old generation.
- ***Parallel Collector***: GC algorithms that can use one or more software threads to perform collection related activities.
- ***Permanent Generation***: Java heap generation where JVM artifacts such as classes and methods are allocated.
- ***Survivor Spaces***: There are two survivor spaces called *from* and *to* survivor spaces. These two alternate their roles every young generation collection and are part of the young generation. Java objects that survive young generation collection continue to be copied into survivor spaces until they are copied enough number of times to exceed the maximum tenuring threshold. After this, the objects are moved to the Old/Tenured generation.
- ***Young Generation***: Java heap generation where object allocation happens. Assuming that more than 90% of objects created by Java programs die young, most of the GC happens in the young generation. Objects that survive young generation collections are copied to survivor spaces until a threshold called maximum tenuring threshold is reached.

---

[i] 62 seconds per data node equals to (62 * 1,000 seconds / 3,600) hours on a 1,000-node cluster. This comes out to a total savings of 17.22 hours of computing cycles.

[ii] 62,000 seconds of computing cycles saved per TeraSort run. Each node consumes 767 seconds of computing cycles per run. This equals (62,000 / 767) nodes; or, 80 nodes.

[iii] 320.3 Watts average power consumed by a non-GC-tuned TeraSort run = 0.3203kWatts = 0.3203 * 8766 hours yearly consumption = 2806.2kWh. Energy cost for one data node = 2806.2kWh * $0.11 = $308.68. Energy cost for 1000 data nodes = $308,680. Best-case annual energy expense savings = 7.45% of $308,680 = $22,996. Average case annual energy expense savings = 2.15% of $308,680 = $6,636. Note: $0.11 per kWh ($0.0011 per Wh) price of electricity assumed. Energy expense savings are under the assumption that all the nodes in the cluster are powered down for the time remaining after performing same amount of work as performed by non-GC-tuned TeraSort run.

Note that the network bottlenecks in bigger cluster environments, such as those with 1,000 nodes, could negatively impact performance and energy savings estimates described here.