Chapter 1
Scaling the Java Virtual Machine on a Many-Core System

Karthik Ganesan, Yao-Min Chen, and Xiaochen Pan

1.1 Introduction

Today, many big data applications use the Java SE platform [13], also called Java Virtual Machine (JVM), as the run-time environment. Examples of such applications include Hadoop Map Reduce [1], Apache Spark [3], and several graph processing platforms [2, 11]. In this chapter, we call these applications the *JVM applications*. Such applications can benefit from modern multicore servers with large memory capacity and the memory bandwidth needed to access it. However, with the enormous amount of data to process, it is still a challenging mission for the JVM platform to scale well with respect to the needs of big data applications. Since the JVM is a multithreaded application, one needs to ensure that the JVM performance can scale well with the number of threads. Therefore, it is important to understand and improve performance and scalability of JVM applications on these multicore systems.

To be able to scale JVM applications most efficiently, the JVM and the various libraries must be scalable across multiple cores/processors and be capable of handling heap sizes that can potentially run into a few hundred gigabytes for some applications. While such scaling can be achieved by scaling-out (multiple JVMs) or scaling-up (single JVM), each approach has its own advantages, disadvantages, and performance implications. Scaling-up, also known as vertical scaling, can be very challenging compared to scaling-out (also known as horizontal scaling), but also has a great potential to be resource efficient and opens up the possibility

K. Ganesan
Oracle Corporation, 5300 Riata Park Court Building A, Austin, TX 78727, USA
e-mail: karthik.ganesan@oracle.com

Y.-M. Chen (✉) • X. Pan
Oracle Corporation, 4180 Network Circle, Santa Clara, CA 95054, USA
e-mail: yaomin.chen@oracle.com; deb.pan@oracle.com

A. Chattopadhyay et al. (eds.), *Emerging Technology and Architecture for Big-data Analytics*, DOI 10.1007/978-3-319-54840-1_1

for features like multi-tenancy. If done correctly, scaling-up usually can achieve higher CPU utilization, putting the servers operating in a more resource and energy efficient state. In this work, we restrict ourselves to the challenges of scaling-up on enterprise-grade systems to provide a focused scope. We elaborate on the various performance bottlenecks that ensue when we try to scale up a single JVM to multiple cores/processors, discuss the potential performance degradation that can come out of these bottlenecks, provide solutions to alleviate these bottlenecks, and evaluate their effectiveness using a representative Java workload.

To facilitate our performance study we have chosen a business analytics workload written in the Java language because Java is one of the most popular programming languages with many existing applications built on it. Optimizing JVM for a representative Java workload would benefit many JVM applications running on the same platform. Towards this purpose, we have selected the LArge Memory Business Data Analytics (LAMBDA) workload. It is derived from the SPECjbb2013 benchmark,[1,2] developed by Standard Performance Evaluation Corporation (SPEC) to measure Java server performance based on the latest features of Java [15]. It is a server side benchmark that models a world-wide supermarket company with multiple point-of-sale stations, multiple suppliers, and a headquarter office which manages customer data. The workload stores all its retail business data in memory (Java heap) without interacting with an external database that stores data on disks. For our study we modify the benchmark in such a way as to scale to very large Java heaps (hundreds of GBs). We condition its run parameter setting so that it will not suffer from an abnormal scaling issue due to inventory depletion.

As an example, Fig. 1.1 shows the throughput performance scaling on our workload as we increase the number of SPARC T5 CPU cores from one to 16.[3] By

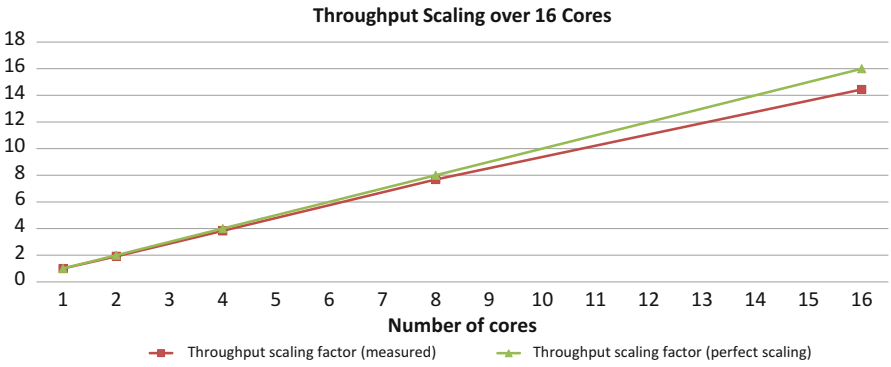


Fig. 1.1 Single JVM scaling on a SPARC T5 server, running the LAMBDA workload

[1]The use of SPECjbb2013 benchmark conforms to SPEC Fair Use Rule [16] for research use.

[2]The SPECjbb2013 benchmark has been retired by SPEC.

[3]Experimental setup for this study is described in Sect. 1.2.3.

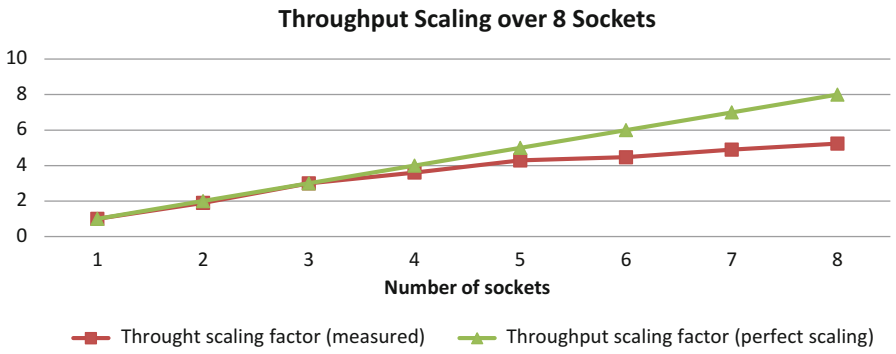


Fig. 1.2 Single JVM scaling on a SPARC M6 server with JDK8 Build 95

contrast, the top ("perfect scaling") curve shows the ideal case where the throughput increases linearly with the number of cores. In reality, there is likely certain system level, OS, Java VM, or application bottleneck to prevent the applications from scaling linearly. And quite often it is a combination of multiple factors that causes the scaling to be non-linear. The main goal of the work described in this chapter is to facilitate application scaling to be as close to linear as possible.

As an example of sub-optimal scaling, Fig. 1.2 shows the throughput performance scaling on our workload as we increase the number of SPARC M6 CPU nsockets from one to eight.[4] There are eight processors ("sockets") on an M6-8 server, and we can run the workload subject to using only the first N sockets. By contrast, the top ("perfect scaling") curve shows the ideal case where the throughput increases linearly with the number of sockets. Below, we discuss briefly the common factors that lead to sub-optimal scaling. We will expand on the key ideas later in this chapter.

1. *Sharing of data objects.* When shared objects that are rarely written to are cached locally, they have the potential to reduce space requirements and increase efficiency. But, the same shared objects can become a bottleneck when being frequently written to, incurring remote memory access latency in the order of hundreds of CPU cycles. Here, a remote memory access can mean accessing the memory not affined to the local CPU, as in a Non-Uniform Memory Access (NUMA) system [5], or accessing a cache that is not affined to the local core, in both cases resulting in a migratory data access pattern [8]. Localized implementations of such shared data objects have proven to be very helpful in improving scalability. A case study that we use to explain this is the concurrent hash map initialization that uses a shared random seed to randomize the layout of hash maps. This shared random seed object causes major synchronization overhead when scaling an application like LAMBDA which creates many transient hash maps.

[4]Experimental setup for this study is described in Sect. 1.2.3.

2. *Application and system software locks.* On large systems with many cores, locks in both user code and system libraries for serialized implementations can be equally lethal in disrupting application scaling. Even standard system calls like malloc in *libc* library tend to have serial portions which are protected by per-process locks. When the same system call is invoked concurrently by multiple threads of same process on a many-core system, these locks around serial portions of implementation become a critical bottleneck. Special implementations of memory allocator libraries like MT hot allocators [18] are available to alleviate such bottlenecks.
3. *Concurrency framework.* Another major challenge involved in scaling is due to inefficient implementations of concurrency frameworks and collection data structures (e.g., concurrent hash maps) using low level Java concurrency control constructs. Utilizing concurrency utilities like JSR166 [10] that provide high quality scalable implementations of concurrent collections and frameworks has a significant potential to improve scalability of applications. One such example is performance improvement of 57% for a workload like LAMBDA derived out of a standard benchmark when using JSR166.
4. *Garbage collection.* As a many-core system is often provisioned with a proportionally large amount of memory, another major challenge in scaling a single JVM on a large enterprise system involves efficiently scaling the Garbage Collection (GC) algorithm to handle huge heap sizes. From our experience, garbage collection pause times (stop-the-world young generation collections) can have a significant effect on the response time of application transactions. These pause times typically tend to be proportional to the nursery size of the Java heap. To reduce the pause times, one solution is to eliminate serial portions of GC phases, parallelizing them to remove such bottlenecks. One such case study includes improvements to the G1 GC [6] to handle large heaps and a parallelized implementation of "*Free Cset*" phase of G1, which has the potential to improve the throughput and response time on a large SPARC system.
5. *NUMA.* The time spent collecting garbage can be compounded due to remote memory accesses on a NUMA based system if the GC algorithm is oblivious to the NUMA characteristics of the system. Within a processor, some cache memories closest to the core can have lower memory access latencies compared to others and similarly across processors of a large enterprise system, some memory banks that are closest to the processor can have lower access latencies compared to remote memory banks. Thus, incorporating the NUMA awareness into the GC algorithm can potentially improve scalability. Most of the scaling bottlenecks that arise out of locks on a large system also tend to become worse on NUMA systems as most of the memory accesses to lock variables end up being remote memory accesses.

The different scalability optimizations discussed in this chapter are accomplished by improving the system software like the Operating System or the Java Virtual Machine instead of changing the application code. The rest of the chapter is

organized as follows: Sect. 1.2 provides the background including the methodologies and tools used in the study and the experimental setup. Section 1.3 addresses the sharing of data objects. Section 1.4 describes the scaling of memory allocators. Section 1.5 expounds on the effective usage of concurrency API. Section 1.6 elaborates on scalable Garbage Collection. Section 1.7 discusses scalability issues in NUMA systems and Sect. 1.8 concludes with future directions.

1.2 Background

The scaling study is often an iterative process as shown in Fig. 1.3. Each iteration consists of four phases: workload characterization, bottleneck identification, performance optimization, and performance evaluation. The goal of each iteration is to remove one or more performance bottlenecks to improve performance. It is an iterative process because a bottleneck may hide other performance issues. When the bottleneck is removed, performance scaling may still be limited by another bottleneck or improvement opportunities which were previously overshadowed by the removed bottleneck.

1. *Workload characterization.* Each iteration starts with characterization using a representative workload. Section 1.2.1 describes selecting a representative workload for this purpose. During workload characterization, performance tools are used in monitoring and capturing key run-time status information and statistics. Performance tools will be described in more detail in Sect. 1.2.2. The result of the characterization is a collection of profiles that can be used in the bottleneck identification phase.
2. *Bottleneck identification.* This phase typically involves modeling, hypothesis testing, and empirical analysis. Here, a bottleneck refers to the cause, or limiting factor, for sub-optimal scaling. The bottleneck often points to, but is not limited to, inefficient process, thread or task synchronization, an inferior algorithm or sub-optimal design and code implementation.
3. *Performance optimization.* Once a bottleneck is identified in the previous phase, in the current phase we try to work out an alternative design or implementation to alleviate the bottleneck. Several possible implementations may be proposed and a comparative study can be conducted to select the best alternative. This phase itself can be an iterative process where several alternatives are evaluated either through analysis or through actual prototyping and subsequent testing.

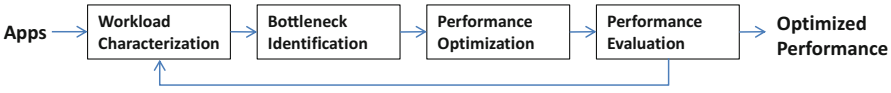


Fig. 1.3 Iterative process for performance scaling: (1) workload characterization, (2) bottleneck identification, (3) performance optimization, and (4) performance evaluation

4. *Performance evaluation.* With the implementation from the performance optimization work in the previous phase, we evaluate whether the performance scaling goal is achieved. If the goal is not yet reached even with the current optimization, we go back to the workload characterization phase and start another iteration.

At each iteration, Amdahl's law [9] is put to practice in the following sense. The goal of many-core scaling is to minimize the serial portion of the execution and maximize the degree of parallelism (DOP) whenever parallel execution is possible. For applications running on enterprise servers, the problem can be solved by resolving issues in the hardware and the software levels. At the hardware level, multiple hardware threads can share an execution pipeline and when a thread is stalled from loading data from memory, other threads can proceed with useful instruction execution in the pipeline. Similarly, at the software level, multiple software threads are mapped to these hardware threads by the operating system in a time-shared fashion. To achieve maximum efficiency, sufficient number of software threads or processes are needed to keep feeding sequences of instructions to ensure that the processing pipelines are busy. A software thread or process being blocked (such as when waiting for a lock) can lead to reduction in parallelism. Similarly, shared hardware resources can potentially reduce parallelism in execution due to hardware constraints. While the problem, as defined above, consists of software-level and hardware-level issues, in this chapter we focus on the software-level issues and consider the hardware micro-architecture as a given constraint to our solution space.

The iterative process continues until the performance scaling goal is reached or adjusted to reflect what is actually feasible.

1.2.1 Workload Selection

In order to expose effectively the scaling bottlenecks of Java libraries and the JVM, one needs to use a Java workload that can scale to multiple processors and large heap sizes from within a single JVM without any inherent scaling problems in the application design. It is also desirable to use a workload that is sensitive to GC pause times as the garbage collector is one of the components that is most difficult to scale when it comes to using large heap sizes and multiple processors. We have found the LAMBDA workload quite suitable for this investigation. The workload implements a usage model based on a world-wide supermarket company with an IT infrastructure that handles a mix of point-of-sale requests, online purchases, and data-mining operations. It exercises modern Java features and other important performance elements, including the latest data formats (XML), communication using compression, and messaging with security. It utilizes features such as the fork-join pool framework and concurrent hash maps, and is very effective in exercising JVM components such as Garbage Collector by tracking response times as small as 10 ms in granularity. It also provides support for virtualization and cloud environments.

The workload is designed to be inherently scalable, both horizontally and vertically using the run modes called *multi-JVM* and *composite* modes respectively. It contains various aspects of e-commerce software, yet no database system is used. As a result, the benchmark is very easy to install and use. The workload produces two final performance metrics: maximum throughput (operations per second) and weighted throughput (operations per second) under response time constraint. Maximum throughput is defined as the maximum achievable injection rate on the System under Test (SUT) until it becomes unsettled. Similarly weighted throughput is defined as the geometric mean of maximum achievable Injection Rates (IR) for a set of response time Service Level Agreements (SLAs) of 10, 50, 100, 200, and 500 ms using the 99th percentile data. The maximum throughput metric is a good measurement of maximum processing capacity, while the weighted throughput gives good indication of the responsiveness of the application running on a server.

1.2.2 Performance Analysis Tools

To study application performance scaling, performance observability tools are needed to illustrate what happens inside a system when running a workload. The performance tools used for our study include Java GC logs, Solaris operating system utilities including cpustat, prstat, mpstat, lockstat, and the Solaris Studio Performance Analyzer.

1. *GC logs.* The logs are very vital in understanding the time spent in garbage collection, allowing us to specify correctly JVM settings targeting the most efficient way to run the workload achieving the least overhead from GC pauses when scaling to multiple cores/processors. An example segment is shown in Fig. 1.4, for the G1 GC [6]. There, we see the breakdown of a stop-the-world (STW) GC event that lasts 0.369 s. The total pause time is divided into four parts: Parallel Time, Code Root Fixup, Clear, and Other. The parallel time represents the time spent in the parallel processing by the 25 GC worker threads. The other parts comprise the serial phase of the STW pause. As seen in the example, Parallel Time and Other are further divided into subcomponents, for which statistics are reported. At the end of the log, we also see the heap occupancy changes from 50.2 GB to 3223 MB. The last line describes that the total user time spent by all GC threads consists of 8.10 s in user land and 0.01 s in the system (kernel), while the elapsed real time is 0.37 s.
2. *cpustat.* The Solaris cpustat [12] utility on SPARC uses hardware counters to provide hardware level profiling information such as cache miss rates, accesses to local/remote memory, and memory bandwidth used. These statistics are invaluable in identifying bottlenecks in the system and ensure that we use the system to the fullest potential. Cpustat provides critical information such as system utilization in terms of cycles per instruction (CPI) and its reciprocal instructions per cycle (IPC) statistics, instruction mix, branch prediction related

```
2016-05-18T16:53:58.019-0700: [GC pause (G1 Evacuation Pause) (young), 0.3690834 secs]
  [Parallel Time: 317.8 ms, GC Workers: 25]
     [GC Worker Start (ms): Min: 333072.4, Avg: 333072.7, Max: 333073.0, Diff: 0.6]
     [Ext Root Scanning (ms): Min: 1.1, Avg: 1.7, Max: 4.4, Diff: 3.3, Sum: 43.1]
     [Update RS (ms): Min: 1.7, Avg: 4.7, Max: 6.7, Diff: 5.0, Sum: 116.8]
        [Processed Buffers: Min: 4, Avg: 14.2, Max: 29, Diff: 25, Sum: 355]
     [Scan RS (ms): Min: 61.3, Avg: 63.5, Max: 64.1, Diff: 2.8, Sum: 1587.1]
     [Object Copy (ms): Min: 246.5, Avg: 246.9, Max: 247.7, Diff: 1.2, Sum: 6172.8]
     [Termination (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.2]
     [GC Worker Other (ms): Min: 0.1, Avg: 0.4, Max: 0.7, Diff: 0.6, Sum: 9.9]
     [GC Worker Total (ms): Min: 316.6, Avg: 317.2, Max: 317.8, Diff: 1.2, Sum: 7929.7]
     [GC Worker End (ms): Min: 333389.6, Avg: 333389.9, Max: 333390.2, Diff: 0.6]
  [Code Root Fixup: 0.0 ms]
  [Clear CT: 7.2 ms]
  [Other: 44.0 ms]
     [Choose CSet: 0.1 ms]
     [Ref Proc: 1.6 ms]
     [Ref Enq: 0.1 ms]
     [Free CSet: 33.6 ms]
  [Eden: 47.0G(47.0G)->0.0B(47.1G) Survivors: 992.0M->960.0M Heap: 50.2G(60.0G)-
>3223.2M(60.0G)]
[Times: user=8.10 sys=0.01, real=0.37 secs]
```

Fig. 1.4 Example of a segment in the Garbage Collector (GC) log showing (1) total GC pause time; (2) time spent in the parallel phase and the number GC worker threads; (3) amounts of time spent in the Code Root Fixup and Clear CT, respectively; (4) amount of time spent in the other part of serial phase; and (5) reduction in heap occupancy due to the GC

```
Section: System Utilization
─────────────────
Stat                            Total

CPI per-core (avg.)             0.64
CPI per-thread (avg.)           5.115
CPI per-core (MIPS est.)        0.73
IPC per-core (avg)              1.56
IPC per-socket (avg)            50.00
Core Util (@select)             79.2%
Core Util (@Instr Cnt)          67.6%
```

Fig. 1.5 An example of cpustat output that shows utilization related statistics. In the figure, we only show the System Utilization section, where CPI, IPC, and Core Utilization are reported

statistics, cache and TLB miss rates, and other memory hierarchy related statistics. Figure 1.5 shows a partial cpustat output that provides system utilization related statistics.

3. *prstat and mpstat.* Solaris prstat and mpstat utilities [12] provide resource utilization and context switch information dynamically to identify phase behavior and time spent in system calls in the workload. This information is very useful in finding bottlenecks in the operating system. Figures 1.6 and 1.7 are examples of a prstat and mpstat output, respectively. The prstat utility looks at resource usage from the process point of view. In Fig. 1.6, it shows that at time instant 2:13:11 the JVM process, with process ID 1472, uses 63 GB of memory, 90% of CPU, and 799 threads while running the workload. However, at time 2:24:33,

```
PID USERNAME  SIZE   RSS STATE   PRI NICE      TIME  CPU PROCESS/NLWP
1472 root      63G   63G cpu4     10    0   2:13:11  90% java/799
```

(a) Normal (non-GC) Phase

```
PID USERNAME  SIZE   RSS STATE   PRI NICE      TIME  CPU PROCESS/NLWP
1472 root      63G   63G cpu2     0    0   2:24:33 5.8% java/475
```

(b) GC Phase

Fig. 1.6 An example of prstat output that shows dynamic process resource usage information. In (**a**), the JVM process (PID 1472) is on cpu4 and uses 90% of the CPU. By contrast, in (**b**) the process goes into GC and uses 5.8% of cpu2

```
SET minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl sze
  0    5   0 3249  4420  335 5459 2574 1232  292    0  4245   85   0   0  14  64
```

(a) vCPUs busy

```
SET minf mjf xcal  intr ithr  csw icsw migr smtx  srw syscl  usr sys  wt idl sze
  0    2   0 2066  3158  192 3715   90  526  321    0  2783   14   0   0  86  64
```

(b) vCPUs idle

Fig. 1.7 An example of mpstat output. In (**a**) we show the dynamic system activities when the processor set (ID 0) is busy. In (**b**) we show the activities when the processor set is fairly idle

the same process has gone into the garbage collection phase, resulting in CPU usage dropped to 5.8% and the number of threads reduced to 475. By contrast, rather than looking at a process, mpstat takes the view from a vCPU (hardware thread) or a set of vCPUs. In Fig. 1.7 the dynamic resource utilization and system activities of a "processor set" is shown. The processor set, with ID 0, consists of 64 vCPUs. The statistics are taken during a sampling interval, typically one second or 5 s. One can contrast the difference in system activities and resource usage taken during a normal running phase (Fig. 1.7a) and during a GC phase (Fig. 1.7b).

4. *lockstat and plockstat.* Lockstat [12] helps us to identify the time spent spinning on system locks and plockstat [12] provides the same information regarding user locks enabling us to understand the scaling overhead that is coming out of spinning on locks. The plockstat utility provides information in three categories: *mutex block*, *mutex spin*, and *mutex unsuccessful spin*. For each category it lists the time (in nanoseconds) in descending order of the locks. Therefore, on the top of the list is the lock that consumes the most time. Figure 1.8 shows an example of plockstat output, where we only extract the lock on the top from each category. For the mutex block category, the lock at address 0x10015ef00 was called 19 times during the capturing interval (1 s for this example). It was

```
Mutex block

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   19    66258 0x10015ef00                  libumem.so.1`umem_cache_alloc+0x50
   31    16040 0x100194e80                  libumem.so.1`umem_cache_alloc+0x50

Mutex spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
  116    38090 0x10015ed00                  libumem.so.1`umem_cache_free+0x68
   92    37586 0x10015ed40                  libumem.so.1`umem_cache_free+0x68

Mutex unsuccessful spin

Count     nsec Lock                         Caller
-------------------------------------------------------------------------------
   49    97353 0x100152e80                  libumem.so.1`umem_cache_alloc+0x50
   42    97960 0x100152d40                  libumem.so.1`umem_cache_alloc+0x50
```

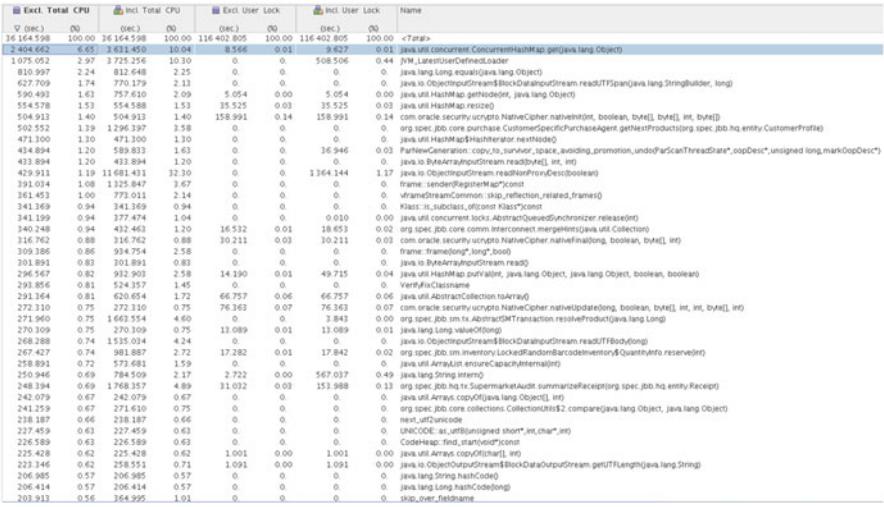
Fig. 1.8 An example of plockstat output, where we show the statistics from three types of locks

called by "libumem.so.1'umem_cache_alloc+0x50" and consumed 66258 ns of CPU time. The locks in the other categories, mutex spin and mutex unsuccessful spin, can be understood similarly.

5. *Solaris studio performance analyzer.* Lastly, Solaris Studio Performance Analyzer [14] provides insights into program execution by showing the most frequently executed functions, caller-callee information along with a timeline view of the dynamic events in the execution. This information about the code is also augmented with hardware counter based profiling information helping to identify bottlenecks in the code. In Fig. 1.9, we show a profile taken while running the LAMBDA workload. From the profile we can identify hot methods that use a lot of CPU time. The hot methods can be further analyzed using the call tree graph, such as the example shown in Fig. 1.10.

1.2.3 Experimental Setup

Two hardware platforms are used in our study. The first is a two-socket system based on the SPARC T5 [7] processor (Fig. 1.11), the fifth generation multicore microprocessor of Oracle's SPARC T-Series family. The processor has a clock frequency of 3.6 GHz, 8 MB of shared last level (L3) cache, and 16 cores where each core has eight hardware threads, providing a total of 128 hardware threads, also known as virtual CPUs (vCPUs), per processor. The SPARC T5-2 system used in our study has two SPARC T5 processors, giving a total of 256 vCPUs available for application use. The SPARC T5-2 server runs Solaris 11 as its operating system. Solaris provides a configuration utility ("psrset") to condition an application to use



Excl. Total CPU		Incl. Total CPU		Excl. User Lock		Incl. User Lock		Name
▽ (sec.)	(%)	(sec.)	(%)	(sec.)	(%)	(sec.)	(%)	
36 164.598	100.00	36 164.598	100.00	116 402.805	100.00	116 402.805	100.00	<Total>
2 404.662	6.65	3 631.450	10.04	8.566	0.01	9.627	0.01	java.util.concurrent.ConcurrentHashMap.get(java.lang.Object)
1 075.052	2.97	3 725.256	10.30	0.	0.	508.506	0.44	JVM_LatestUserDefinedLoader
810.397	2.24	812.648	2.25	0.	0.	0.	0.	java.lang.Long.equals(java.lang.Object)
627.709	1.74	770.179	2.13	0.	0.	0.	0.	java.io.ObjectInputStream$BlockDataInputStream.readUTFSpan(java.lang.StringBuilder, long)
590.493	1.63	757.610	2.09	5.054	0.00	5.054	0.00	java.util.HashMap.getNode(int, java.lang.Object)
554.578	1.53	554.588	1.53	35.525	0.03	35.525	0.03	java.util.HashMap.resize()
504.913	1.40	504.913	1.40	158.991	0.14	158.991	0.14	com.oracle.security.ucrypto.NativeCipher.nativeInit(int, boolean, byte[], byte[], int, byte[])
502.552	1.39	1296.397	3.58	0.	0.	0.	0.	org.spec.jbb.core.purchase.CustomerSpecificPurchaseAgent.getNextProducts(org.spec.jbb.hq.entity.CustomerProfile)
471.100	1.30	471.100	1.30	0.	0.	0.	0.	java.util.HashMap$HashIterator.nextNode()
434.894	1.20	589.833	1.63	0.	0.	36.946	0.03	ParNewGeneration::copy_to_survivor_space_avoiding_promotion_undo(ParScanThreadState*,oopDesc*,unsigned long,markOopDesc*)
433.894	1.20	433.894	1.20	0.	0.	0.	0.	java.io.ByteArrayInputStream.read(byte[], int, int)
429.911	1.19	11 681.431	32.30	0.	0.	1 164.144	1.17	java.io.ObjectInputStream.readNonProxyDesc(boolean)
391.014	1.08	1 125.847	3.67	0.	0.	0.	0.	frame::sender(RegisterMap*)const
361.453	1.00	771.011	2.14	0.	0.	0.	0.	vframeStreamCommon::skip_reflection_related_frames()
341.369	0.94	341.369	0.94	0.	0.	0.	0.	Klass::is_subclass_of(const Klass*)const
341.199	0.94	377.474	1.04	0.	0.	0.010	0.00	java.util.concurrent.locks.AbstractQueuedSynchronizer.release(int)
340.248	0.94	432.463	1.20	16.532	0.01	18.653	0.02	org.spec.jbb.core.comm.Interconnect.mergeHints(java.util.Collection)
316.762	0.88	316.762	0.88	30.211	0.03	30.211	0.03	com.oracle.security.ucrypto.NativeCipher.nativeFinal(long, boolean, byte[], int)
309.186	0.86	934.754	2.58	0.	0.	0.	0.	frame::frame(long*,long*,bool)
301.891	0.83	301.891	0.83	0.	0.	0.	0.	java.io.ByteArrayInputStream.read()
296.567	0.82	932.903	2.58	14.190	0.01	49.715	0.04	java.util.HashMap.putVal(int, java.lang.Object, java.lang.Object, boolean, boolean)
293.856	0.81	524.157	1.45	0.	0.	0.	0.	VerifyFix:Classname
291.364	0.81	620.654	1.72	66.757	0.06	66.757	0.06	java.util.AbstractCollection.toArray()
272.110	0.75	272.110	0.75	76.363	0.07	76.363	0.07	com.oracle.security.ucrypto.NativeCipher.nativeUpdate(long, boolean, byte[], int, int, byte[], int)
271.960	0.75	1 663.554	4.60	0.	0.	3.843	0.00	org.spec.jbb.sm.tx.AbstractSMTransaction.resolveProduct(java.lang.Long)
270.109	0.75	270.109	0.75	13.089	0.01	13.089	0.01	java.lang.Long.valueOf(long)
268.288	0.74	1 535.034	4.24	0.	0.	0.	0.	java.io.ObjectInputStream$BlockDataInputStream.readUTFBody(long)
267.427	0.74	981.887	2.72	17.282	0.01	17.842	0.02	org.spec.jbb.sm.inventory.LockedRandomBarcodeInventory$QuantityInfo.reserve(int)
258.891	0.72	573.681	1.59	0.	0.	0.	0.	java.util.ArrayList.ensureCapacityInternal(int)
250.946	0.69	784.509	2.17	2.722	0.00	567.037	0.49	java.lang.String.intern()
248.394	0.69	1 768.357	4.89	31.032	0.03	153.988	0.13	org.spec.jbb.hq.tx.SupermarketAudit.summarizeReceipt(org.spec.jbb.hq.entity.Receipt)
242.079	0.67	242.079	0.67	0.	0.	0.	0.	java.util.Arrays.copyOf(java.lang.Object[], int)
241.259	0.67	271.610	0.75	0.	0.	0.	0.	org.spec.jbb.core.collections.CollectionUtils$2.compare(java.lang.Object, java.lang.Object)
238.187	0.66	238.187	0.66	0.	0.	0.	0.	next_utf2unicode
227.459	0.63	227.459	0.63	0.	0.	0.	0.	UNICODE::as_utf8(unsigned short*,int,char*,int)
226.589	0.63	226.589	0.63	0.	0.	0.	0.	CodeHeap::find_start(void*)const
225.428	0.62	225.428	0.62	1.001	0.00	1.001	0.00	java.util.Arrays.copyOf(char[], int)
223.346	0.62	258.551	0.71	1.091	0.00	1.091	0.00	java.io.ObjectOutputStream$BlockDataOutputStream.getUTFLength(java.lang.String)
206.985	0.57	206.985	0.57	0.	0.	0.	0.	java.lang.String.hashCode()
206.414	0.57	206.414	0.57	0.	0.	0.	0.	java.lang.Long.hashCode(long)
203.913	0.56	364.995	1.01	0.	0.	0.	0.	skip_over_fieldname

Fig. 1.9 An example of Oracle Solaris Studio Performer Analyzer profile, where we show the methods ranked by exclusive cpu time

Fig. 1.10 An example of Oracle Solaris Studio Performer Analyzer call tree graph

only a subset of vCPUs. Our experimental setup includes running the LAMBDA workload on configurations of 1 core (8 vCPUs), 2 cores (16 vCPUs), 4 cores (32 vCPUs), 8 cores (64 vCPUs), 1 socket (16 cores/128 vCPUs), and 2 sockets (32 cores/256 vCPUs).

The second hardware platform is an eight-socket SPARC M6-8 system that is based on the SPARC M6 [17] processor (Fig. 1.12). The SPARC M6 processor has a clock frequency of 3.6 GHz, 48 MB of L3 cache, and 12 cores. Same as SPARC T5, each M6 core has eight hardware threads. This gives a total of 96 vCPUs per

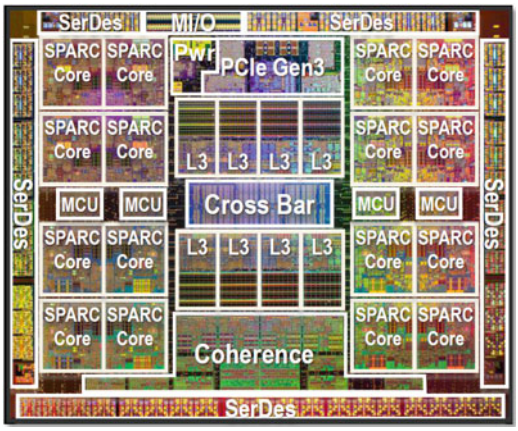


Fig. 1.11 SPARC T5 processor [7]

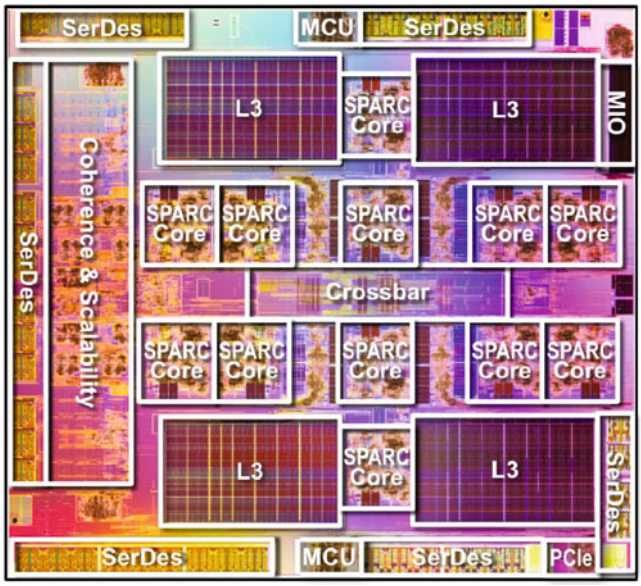


Fig. 1.12 SPARC M6 processor [17]

processor socket, for a total of 768 vCPUs for the full M6-8 system. The SPARC M6-8 server runs Solaris 11. Our setup includes running the LAMBDA workload on configurations of 1 socket (12 cores/96 vCPUs), 2 sockets (24 cores/192 vCPUs), 4 sockets (48 cores/384 vCPUs), and 8 sockets (96 cores/384 vCPUs).

Several JDK versions have been used in the study. We will call out the specific versions in the sections to follow.

1.3 Thread-Local Data Objects

A globally shared data object when protected by locks on the critical path of application leads to the serial part of Amdahl's law. This causes less than perfect scaling. To improve degree of parallelism, the strategy is to "unshare" such data objects that cannot be efficiently shared. Whenever possible, we try to use data objects that are local to the thread, and not shared with other threads. This can be more subtle than it sounds, as the following case study demonstrates.

Hash map is a frequently used data structure in Java programming. To minimize the probability of collision in hashing, JDK 7u6 introduced an alternative hash map implementation that adds randomness in the initiation of each *HashMap* object. More precisely, the alternative hashing introduced in JDK 7u6 includes a feature to randomize the layout of individual map instances. This is accomplished by generating a random mask value per hash map. However, the implementation in JDK 7u6 uses a *shared* random seed to randomize the layout of hash maps. This shared random seed object causes significant synchronization overhead when scaling an application like LAMBDA which creates many transient hash maps during the run. Using Solaris Studio Analyzer profiles, we observed that for an experiment run with 48 cores of M6, CPUs were saturated and 97% of CPU time was spent in the *java.util.Random.nextInt()* function achieving less than 15% of the system's projected performance. The problem came out of *java.util.Random.nextInt()* updating global state, causing synchronization overhead as shown in Fig. 1.13.

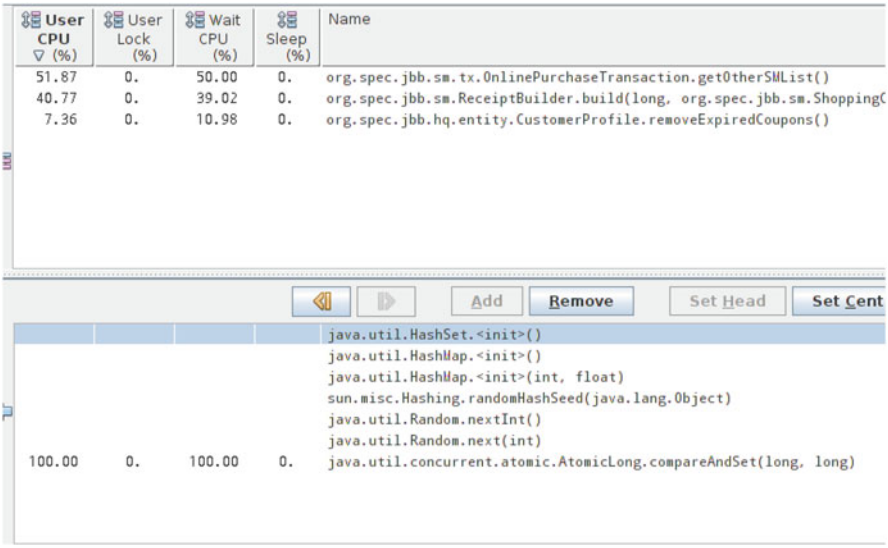


Fig. 1.13 Scaling bottleneck due to *java.util.Random.nextInt*

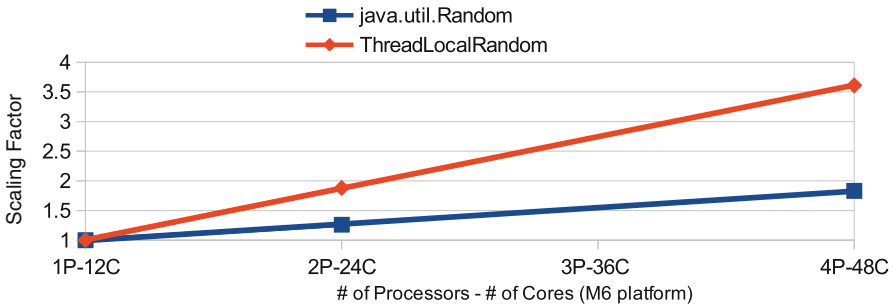


Fig. 1.14 LAMBDA Scaling with *ThreadLocalRandom* on M6 platform

The OpenJDK bug JDK-8006593 tracks the aforementioned issue and uses a thread-local random number generator, *ThreadLocalRandom* to resolve the problem, thereby eliminating the synchronization overhead and improving performance of the LAMBDA workload significantly. When using the *ThreadLocalRandom* class, a generated random number is isolated to the current thread. In particular, the random number generator is initialized with an internally generated seed. In Fig. 1.14, we can see that the 1-to-4 processor scaling improved significantly from a scaling factor of 1.83 (when using *java.util.Random*) to 3.61 (when using *java.util.concurrent.ThreadLocalRandom*). The same performance fix improves the performance of a 96-core 8-processor large M6 system by 4.26 times.

1.4 Memory Allocators

Many in-memory business data analytics applications allocate and deallocate memory frequently. While Java uses an internal heap and most of the allocations happen within this heap, there are components of applications that end up allocating outside the Java heap using native memory allocators provided by the operating system. One such commonly seen component would be native code, which are code parts written specific to a hardware and operating system platform accessed using the Java Native Interface. Native code uses system *malloc()* to dynamically allocate memory. Many business analytics applications use crypto functionality for security purposes and most of the implementations for crypto functions are hand optimized native code which allocates memory outside the Java heap. Similarly, network I/O components are also frequently implemented to allocate and access memory outside the Java heap. In business analytics applications, we see many such crypto and network I/O functions used regularly resulting in calls to the OS system call *malloc()* from within the JVM.

Most modern operating systems, like Solaris, have a heap segment, which allows for dynamic allocation of space during run time using system calls such as *malloc()*. When such a previously allocated object is deallocated, the space used by the object

can be reused. For the most efficient allocation and reuse of space, the solution is to maintain a heap inventory (alloc/free list) stored in a set of data structures in the process address space. In this way, calling *free()* does not return the memory back to the system; it is put in the free-list. The traditional implementation (such as the default memory allocator in libc) protects the entire inventory using a single per-process lock. Calls to memory allocation and de-allocation routines manipulate this set of data structures while holding the lock. This single lock causes a potential performance bottleneck when we scale a single JVM to a large number of cores and the target Java application has *malloc()* calls from components like network I/O or crypto. When we profiled the LAMBDA workload using Solaris Studio Analyzer, we found that the *malloc()* calls were showing higher than expected CPU time. A further investigation using the lockstat and plockstat tools revealed a highly contended lock called the depot lock. The depot lock protects the heap inventory of free pages. This motivated us to explore scalable implementations of memory allocators.

A set of newer memory allocators, called Multi-Thread (MT) Hot allocators [18], partition the inventory and the associated locks into arrays to reduce the contention on the inventory. A value derived from the caller's CPU ID is used as an index into the array. It is worth noting that a slight side effect of this approach is that it can cause more memory usage. This happens because instead of a single free-list of memory, we now have a disjoint set of free-lists. This tends to require more space since we will have to ensure each free-list has sufficient memory to avoid run-time allocation failures.

The *libumem* [4] memory allocator is an MT-Hot allocator included in Solaris. To evaluate the improvement from this allocator, we use the LD_PRELOAD environment variable to preload this library, there by *malloc()* implementation in this library is used over the default implementation in the libc library. The improvement in performance seen when using libumem over libc is shown in Fig. 1.15. With the MT-hot allocator, the performance in terms of throughput increases by 106%, 213%, and 478% for 8-core (half processor), 16-core (1 processor), and 32-core

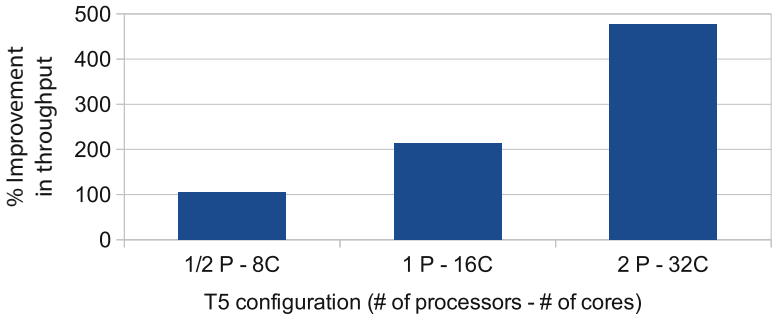


Fig. 1.15 LAMBDA workload throughput improvement with MT-hot alloc over libc *malloc()* on T5-2

(2 processors) configurations, respectively, on T5-2 in comparison to *malloc()* in libc. Note that while JVM uses *mmap()*, instead of *malloc()*, for allocation of its garbage-collectable heap region, the JNI part of JVM does use *malloc()*, especially for the crypto and security related processing. The workload LAMBDA has a significant part of operation in crypto and security, so the effect of MT Hot allocator is quite significant. After switching to an MT-Hot allocator, the hottest observed lock "depot lock" in the memory allocator disappeared and reduced the time spent in locks by a factor of 21. This confirmed the necessity of an MT-Hot memory allocator for successful scaling.

1.5 Java Concurrency API

Ever since JDK 1.2, Java has included a standard set of collection classes called the Java *collections* framework. A collection is an object that represents a group of objects. Some of the fundamental and popularly used collections are dynamic arrays, linked lists, trees, queues, and hashtables. The collections framework is a unified architecture that enables storage and manipulation of the collections in a standard way, independent of underlying implementation details. Some of the benefits of the collections framework include reduced programming effort by providing data structures and algorithms for programmers to use, increased quality from high performance implementation and enabling reusability and interoperability. The collection framework is used extensively in almost every Java program these days. While these pre-implemented collections make the job of writing single threaded application so much easier, writing concurrent multithreaded programs is still a difficult job. Java provided low level threading primitives such as synchronized blocks, Object.wait and Object.notify, but these were too fine grained facilities forcing programmers to implement high level concurrency primitives, which are tediously hard to implement correctly and often were non-performant.

Later, a concurrency package, comprising several concurrency primitives and many collection-related classes, as part of the JSR 166 [10] library, was developed. The library was aimed at providing high quality implementation of classes to include atomic variables, special-purpose locks, barriers, semaphores, high performant threading utilities like thread pools and various core collections like queues and hashmaps designed and optimized for multithreaded programming. The concurrency APIs developed by the JSR 166 working group were included as part of the JDK 5.0. Since then both Java SE 6 and Java SE 7 releases introduced updated versions of the JSR 166 APIs as well as several new additional APIs. Availability of this library relieves the programmer from redundantly crafting these utilities by hand, similar to what the collections framework did for data structures. Our early evaluation of Java SE 7 found a major challenge in scaling from the implementations of concurrent collection data structures (such as concurrent hash maps) using low level Java concurrency control constructs. We explored utilizing concurrency utilities from JSR 166, leveraging the scalable implementations of

concurrent collections and frameworks and saw very significant improvement in the scalability of applications. Specifically, the LAMBDA workload code uses the Java class *java.util.concurrent.ConcurrentHashMap*. The efficiency of its underlying implementation affects performance quite significantly. For example, comparing the *ConcurrentHashMap* implementation of JDK8 over JDK7, there is an improvement of about 57% in throughput due to the improved JSR 166 implementation.

1.6 Garbage Collection

Automatic Garbage Collection (GC) is the cornerstone of memory management in Java enabling developers to allocate new objects without worrying about deallocation. The Garbage Collector reclaims memory for reuse ensuring that there are no memory leaks and also provides security from vulnerabilities in terms of memory safety. But, automatic garbage collection comes at a small performance cost for resolving these memory management issues. It is an important aspect of real world enterprise application performance, as GC pause times translate into unresponsiveness of an application. Shorter GC pauses will help the applications to meet more stringent response time requirements. When heap sizes run into a few hundred gigabytes on contemporary many-core servers, achieving low pause times require the GC algorithm to scale efficiently with the number of cores. Even when an application and the various dependent libraries are ensured to scale well without any bottlenecks, it is important that the GC algorithm also scales well to achieve scalable performance.

It may be intuitive to think that the garbage collector will identify and eliminate dead objects. But, in reality it is more appropriate to say that the garbage collector rather tracks the various live objects and copies them out, so that the remaining space can be reclaimed. The reason that such an implementation is preferred in the modern collectors is that, most of the objects die young and it is much faster to copy the fewer remaining live objects out than tracking and reclaiming the space of each of the dead objects. This will also give us a chance to compact the remaining live objects ensuring a defragmented memory. Modern garbage collectors have a generational approach to this problem, maintaining two or more allocation regions (generations) with objects grouped into these regions based on their age. For example, the G1 GC [6] reduces heap fragmentation by incremental parallel copying of live objects from one or more sets of regions (called Collection Set or CSet in short) into different new region(s) to achieve compaction. The G1 GC [6] tracks references into regions using independent Remembered Sets (RSets). These RSets enable parallel and independent collection of these regions because each region's RSet can be scanned independently for references into that region as opposed to scanning the entire heap. The G1 GC has a multiphase complex algorithm that has both parallel and serial code components contributing to Stop The World (STW) evacuation pauses and concurrent collection cycles.

With respect to the LAMBDA workload, pauses due to GC directly affect the response time metric monitored by the benchmark. If the GC algorithm does not scale well, long pauses will exceed the latency requirements of the benchmark resulting in lower throughput. In our experiments with monitoring the LAMBDA workload on an M6 server, we had some interesting observations. While at the regular throughput phase of the benchmark run, the system CPUs were fully utilized almost at 100%. By contrast, there was much more CPU headroom (75%) during a GC phase, hinting at possible serial bottlenecks in Garbage Collection. By collecting and analyzing code profiles using Solaris Studio Analyzer, the time the worker threads of the LAMBDA workload spend waiting on conditional variables increase from 3%, for a 12-core (single-processor) run, to to 16%, for a 96-core (8-processor) run on M6. This time was mostly spent in lwp_cond_wait() waiting for the young generation stop-the-world garbage collection, observed to be in sync with the GC events based on a visual timeline review of Studio Analyzer profiles. Further the call stack of the worker threads consists of the *SafepointSynchronize::block()* function consuming 72% of time clearly pointing at the scalability issue in garbage collection.

G1 GC [6] provides a breakdown of the time spent in various phases to the user via verbose GC logs. Analyzing these logs pointed to a major serial component "*Free Cset*," for which the processing time was proportional to the size of the heap (mainly the nursery component responsible for the storage of the young objects). This particular phase of the GC algorithm was not parallelized and some of the considerations included the cost involved in thread creation for parallel execution. While thread creation may be a major overhead and an overkill for small heaps, such a cost can be amortized if the heap size is large and running into hundreds of gigabytes. A parallelized implementation of the "*Free Cset*" phase was created for testing purposes as part of the JDK bug JDK-8034842. We noticed that this parallelized implementation for the "*Free Cset*" phase of G1 GC provided major reduction in pause times for this phase for the LAMBDA workload. The pause times for this phase went down from 230 ms to 37 ms for scaled runs on 8 processors (96 cores) of M6. The ongoing work in fully parallelizing the *FreeCset* phase is tracked in the JDK bug report JDK-8034842. Also, we observed that a major part of the scaling overhead that came out of garbage collection on large many-core systems was from accesses to remote memory banks in a Non-Uniform Memory Access (NUMA) system. We examine this impact further in the following subsection.

1.7 Non-uniform Memory Access (NUMA)

Most of the modern many-core systems are shared memory systems that have Non-Uniform Memory Access (NUMA) latencies. Modern operating systems like Solaris have memory (DRAM, cache) banks and CPUs classified into a hierarchy of locality groups (lgroup). Each lgroup includes a set of CPU and memory banks, where the leaf lgroups include the CPUs and memory banks that are closest to each other in

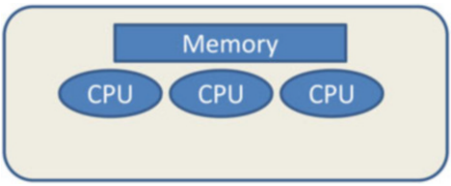


Fig. 1.16 Machine with single latency is represented by only one lgroup

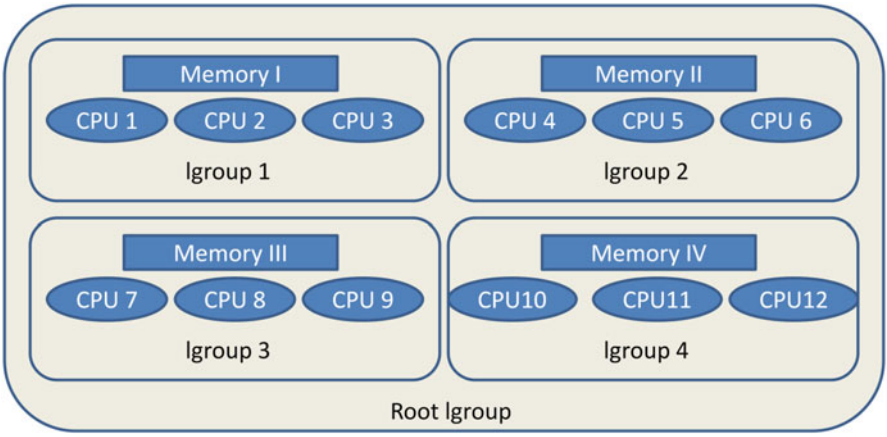


Fig. 1.17 Machine with multiple latency is represented by multiple lgroups

terms of access latency, with the hierarchy being organized similarly up to the root. Figure 1.16 shows a typical system with a single memory latency, represented by one lgroup. Figure 1.17 shows a system with multiple memory latencies, represented by multiple lgroups. In this organization, the CPUs 1–3 belong to lgroup1 and will have the least latency to access Memory I. Similarly, CPUs 4–6 to Memory II, CPUs 7–9 to Memory III, and CPUs 10–12 to Memory IV will have the least local access latencies. When a CPU accesses a memory location that is outside its local lgroup, a longer remote memory access latency will be incurred.

In systems with multiple lgroups, it would be most desirable to have the data that is being accessed by the CPUs in their nearest lgroups, thus incurring shortest access latencies. Due to high remote memory access latency, it is very important that the operating system be aware of the NUMA characteristics of the underlying hardware. Additionally, it is a major value add if the Garbage Collector in the Java Virtual Machine is also engineered to take these characteristics into account. For example, the initial allocation of space for each thread can be made so that it is in the same lgroup as that of the CPU on which the thread is running. Secondly, the GC algorithm can also make sure that when data is compacted or copied from one generation to another, some preference can be given to ensure that the data is not copied to a remote lgroup with respect to the thread that is most frequently accessing the data. This will enable easier scaling across multiple cores and multiple processors of large enterprise systems.

To understand the impact of remote memory accesses on the performance of garbage collector and the application, we profiled the LAMBDA workload with the help of pmap and Solaris tools cpustat and busstat, breaking down the distribution of heap/stack to various lgroups. The Solaris tool pmap provides a snapshot of process data at a given point of time in terms of the number of pages, size of pages, and the lgroup in which the pages are resident. This can be used to get a spatial breakdown of the Java heap to various lgroups. The utility cpustat on SPARC uses hardware counters to provide hardware level profiling information such as cache miss rates and access latencies to local and remote memory banks. Similarly, the busstat utility provides memory bandwidth usage information, again broken down at memory bank/lgroup granularity. Our initial set of observations using pmap showed that the heap was not distributed uniformly across the different lgroups and that a few lgroups were used more frequently than the rest. Cpustat and bustat information corroborated this observation, showing high access latencies and bandwidth usage for these stressed set of lgroups.

To alleviate this, we tried using key JVM flags which provide hints to the GC algorithm about memory locality. First, we found that the usage of the flag *-XX:+UseNUMAInterleaving* can be indispensable in hinting to the JVM to distribute the heap equally across different lgroups and avoid bottlenecks that will arise from data being concentrated on a few lgroups. While *-XX:+UseNUMAInterleaving* will only avoid concentration of data in particular banks, flags like *-XX:+UseNUMA* when used with Parallel Old Garbage Collector have the potential to tailor the algorithm to be aware of NUMA characteristics and increase locality. Further, operating system flags like *lpg_alloc_prefer* in Solaris 11 and *lgrp_mem_pset_aware* in Solaris 12, when set to true, hint to the OS to allocate large pages in the local lgroup rather than allocating them in a remote lgroup. This can be very effective in improving memory locality in scaled runs. The *lpg_alloc_prefer* flag, when set to true can increase the throughput of the LAMBDA workload by about 65% on the M6 platform, showing the importance of data locality. While ParallelOld is an effective stop-the-world collector, concurrent garbage collectors like CMS and G1 GC [6] are most useful in real world response time critical application deployments. The enhancement requests that track the implementation of NUMA awareness into G1 GC and CMS GC are JDK-7005859 and JDK-6468290.

1.8 Conclusion and Future Directions

We present an iterative process for performance scaling JVM applications on many-core enterprise servers. This process consists of workload characterization, bottleneck identification, performance optimization, and performance evaluation in each iteration. As part of workload characterization, we first provide an overview of the various tools that are provided as part of modern operating systems most useful to profile the execution of workloads. We use a data analytics workload, LAMBDA as an example to explain the process of performance scaling. We identify

various bottlenecks in scaling this application such as synchronization overhead due to shared objects, serial resource bottleneck in memory allocation, lack of usage of high level concurrency primitives, serial implementations of Garbage Collection phases, and uneven distribution of heap on a NUMA machine oblivious to the NUMA characteristics by using the profiled data. We further discuss in depth the root cause of each bottleneck and present solutions to address them. These solutions include unsharing of shared objects, usage of multicore friendly allocators such as MT-Hot allocators, high performance concurrency constructs as in JSR166, parallelized implementation of Garbage Collection phases, and NUMA aware garbage collection. Taken together, the overall improvement for the proposed solutions is more than 16 times on an M6-8 server for the LAMBDA workload in terms of maximum throughput.

Future directions include hardware accelerations to address scaling bottlenecks, increased emphasis on the response time metric where GC performance and scalability will be a key factor, and horizontal scaling aspects of big data analytics where disk and network I/O will play crucial roles.

Acknowledgements We would like to thank Jan-Lung Sung, Pallab Bhattacharya, Staffan Friberg, and other anonymous reviewers for their valuable feedback to improve the chapter.

References

1. Apache, Apache Hadoop (2017). Available: https://hadoop.apache.org
2. Apache Software Foundation, Apache Giraph (2016). Available https://giraph.apache.org
3. Apache Spark (2017). Available https://spark.apache.org
4. Oracle, Analyzing Memory Leaks Using the libumem Library [online]. https://docs.oracle.com/cd/E19626-01/820-2496/geogv/index.html
5. W. Bolosky, R. Fitzgerald, M. Scott, Simple but effective techniques for numa memory management. SIGOPS Oper. Syst. Rev. **23**(5), 19–31 (1989)
6. D. Detlefs, C. Flood, S. Heller, T. Printezis, Garbage-first garbage collection, in *Proceedings of the 4th International Symposium on Memory Management* (2004), pp. 37–48
7. J. Feehrer, S. Jairath, P. Loewenstein, R. Sivaramakrishnan, D. Smentek, S. Turullols, A. Vahidsafa, The Oracle Sparc T5 16-core processor scales to eight sockets. IEEE Micro **33**(2), 48–57 (2013)
8. K. Ganesan, L.K. John, Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. IEEE Trans. Comput. **63**(4), 833–846 (2014)
9. M.D. Hill, M.R. Marty, Amdahl's law in the multicore era. Computer **41**(07), 33–38 (2008)
10. D. Lea, Concurrency JSR-166 interest site (2014). http://gee.cs.oswego.edu/dl/concurrency-interest/
11. Neo4j, Neo4j graph database (2017). Available https://neo4j.com
12. Oracle, Man pages section 1M: system Administration Commands (2016). [Online]. Available http://www.oracle.com
13. Oracle Corporation, Java SE platform (2017). Available http://www.oracle.com/technetwork/java/javase/overview/index.html
14. Oracle solaris studio performance analyzer (2014). http://docs.oracle.com/cd/E18659_01/html/821\discretionary-1379/

15. C. Pogue, A. Kumar, D. Tollefson, S. Realmuto, Specjbb2013 1.0: an overview, in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering* (2014), pp. 231–232
16. Standard Performance Evaluation Corporation, Spec fair use rule. academic/research usage (2015). [Online]. Available http://www.spec.org/fairuse.html#Academic
17. A. Vahidsafa, S. Bhutani, SPARC M6 oracle's next generation processor for enterprise systems, in *Hotchips 25* (2013). [Online]. Available http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.90-Processors3-epub/HC25.27.920-SPARC-M6-Vahidsafa-Oracle.pdf
18. R.C. Weisner, How memory allocation affects performance in multithreaded programs (2012). http://www.oracle.com/technetwork/articles/servers-storage-dev/mem\discretionary-alloc\discretionary-1557798.html