Parallel GC

Presented by Eleanor Ainy
16/12/2014

Chapter Summary:

This chapter describes how to use parallelism of collector threads in each of the 4 components of tracing GC - marking, copying, sweeping and compaction. The goal is to reduce time overhead of GC and pause times.

Each solution to this problem deals with the challenges of load balancing (how to distribute the work & resources) and synchronization between the different collection threads. Synchronization is needed for correctness and to avoid repeating work, but has time and space overheads. The algorithms described try to minimize the coordination cost by using thread-local data structures. They mostly over-partition the work into smaller tasks and have threads compete to claim one task at a time to execute.

The chapter differentiates between processor-centric algorithms and memory-centric algorithms. In processor-centric algorithms, threads usually steal work that varies in size from other threads and there is little regard to the location of the objects. In memory-centric algorithms, however, location is taken into greater account. Moreover, the threads in these algorithms acquire fixed size work from shared pools and operate on continuous blocks of heap memory.

Almost all algorithms describes in this chapter have the same form - a loop of acquire work, perform work and generate work, that ends once the termination detector detects termination. First, the chapter describes several parallel marking algorithms. In the first (Endo et al) algorithm each marker thread has its own local mark stack and a stealable work queue and locks are used for synchronization. Next, the chapter describes the Food et al algorithm. Each thread has a fixed size stealable deque. All threads share a global overflow set. In the perform work phase, a thread pops work from its deque and tries to push new work to its deque, but if an overflow occurs then half of the deque (including the new work) is transferred to the overflow set. The chapter also shows Wu and Li's Parallel Tracing algorithm using channels. Threads exchange marking tasks through single writer, single reader channels. If the thread's stack is empty, it takes a task from some input channel <k -->me>. When a thread generates a new task, it first checks whether any other thread k needs work. If so, adds the task to the output channel <me --> k>. Otherwise, pushes the task to its own stack.

The next algorithms the chapter describes are copying algorithms. In Cheng and Belollch algorithm, Each copying thread is given its own stack and transfers work between its local stack and a shared stack. Using rooms, they allow multiple threads to: pop elements from the shared stack in parallel and push elements to the shared stack in parallel, but not pop and push in parallel. The chapter then presents the copying algorithm using block- structured heaps where the heap is divided into chunks and each chink is divided into blocks. Threads compete to acquire scan blocks. To reduce contention the algorithm in some cases prevents from threads to acquire new scan blocks

(depending on the location of the scan pointer with regards to the object and the size of the object).

In addition, Endo et al [1997] Lazy Sweeping algorithm is describes. Finally, parallel compaction algorithm are presented. Rather than sliding all live data to one end of the heap space, like uniprocessor compaction algorithms, the first algorithm that I chose to describe divides the heap space into several regions, one for each compacting thread. To reduce fragmentation, threads alternate the direction in which they move objects in even and odd numbered regions. The threads obtain heap units and count the volume of live data in their units. This is done in order to divide the heap into regions that contain roughly the same amount of live data. The threads then install forwarding addresses in each unit update references. Finally, each thread moves objects in its oen obtained region. The second compaction algorithm uses only 1 pass over the heap instead of 3 passes and reduces fragmentation.

Contributions (things that weren't present in the chapter itself):

- A video lesson https://www.youtube.com/watch?v=YhKZe22tZlc that explains about parallel GC in Java and contains illustrations of the different existing variations. According to the video, there are two variations of parallel GC in JVM. The first variation consists of multi-threaded young generation (minor) GC with **single** threaded old generation GC and is enabled using a command line option UseParallelGC. The second variation consists of a multi-threaded young generation (minor) GC with multi-threaded old generation GC, and is enabled using UseParallelOldGC command line option. Parallel GC is the default GC for server class machines (machines that have at least 2 virtual processors and at least 2 GB of memory).

- Three useful blog posts:
  1) http://www.cubrid.org/blog/dev-platform/understanding-java-garbage-collection/
  Presents the processes for different GC algorithms, how GC works, what Young and Old Generation is, the 5 types of GC in JDK 7, and the performance implications are for each of these GC types.
  2) http://www.cubrid.org/blog/dev-platform/how-to-monitor-java-garbage-collection/
  Presents how JVM actually runs the Garbage Collection in the real time, how we can monitor GC, and which tools we can use to make this process faster and more effective.
  3) http://www.cubrid.org/blog/dev-platform/how-to-tune-java-garbage-collection/
  Presents best options for GC tuning with examples of possible tuning.

During the discussion after the presentation the following points came up:

- The tradeoff in the choice of the chunk size in parallel copying: on the one hand, the chunks should be large in order to reduce contention on the chunk manager. On the other hand, large chunks do not offer an appropriate granularity for balancing the load so they should be broken into smaller blocks which can act as the work quanta.
- Parallel copying with no synchronization can cause issues. For example if an object is copied twice by two different threads, space would be wasted and in the worst case the two replicas might be updated later with conflicting values.