New Parallel Full GC

Problem

The current implementation of the Parallel Full GC is too rigid for Lilliput 2's Compact Identity Hash-Code. Specifically, it does not allow for objects to be resized/expanded when they move, which is a requirement for Compact Identity Hash-Code. The reason for that lies in the fact that the algorithm parallelizes by dividing up the heap into equal-sized regions (with overlapping objects), and pre-compute destination boundaries for each region by inspecting the size of each live object per region. However, we can't determine the size of objects until we know whether or not an object will move at all. This is further complicated by the fact that we cannot even assume that only a dense prefix will not move - the expansion of moved objects can lead to a situation where a subsequent object would not move.

Proposed new Algorithm

The basic idea is to not make assumptions about object sizes, and instead determine the destination location more dynamically. We can adapt the algorithm that is used by G1 and Shenandoah GC. The difficulty is that in G1 and Shenandoah, regions are a bit more rigid in that they don't allow objects that cross region boundaries. That property makes parallelization much easier because worker threads can fully own a region without potential interference from other worker threads.

More flexible region sizes

Therefore we need to make regions of more flexible sizes. In the (single-threaded) summary phase that follows after marking and precedes forwarding /adjusting-ptrs/compaction, we set up our list of regions by starting out with equal-sized regions, and then adjusting each region's bottom upwards to be the first word of the region that is not an overlapping object, and adjust its end upwards to the first word that is not an overlapping object (which will also be the bottom of the subsequent region).

Forwarding Phase

With those more flexible regions set-up, we can basically 1:1 adapt G1/Shenandoah's algorithm for the forwarding and compaction phases. Forwarding works like this:

- From the global list of regions, workers atomically claim regions serially. The first claimed regions becomes that workers current source and destination region. Later, source and destination are likely to become different regions. As the names imply, source is where we compact from, and destination is where we compact to. The destination region maintains the current compact-point, which initially is the destination region's bottom. The worker also maintains a list of destination regions. Claimed regions also get appended to the tail of the destination-region-list, that is they may become compaction destinations, once the current compaction destination is exhausted.
- The worker then scans the current source region's live objects. Each live object gets assigned a forwarding address, which is the current compact point. The compact point is then advanced by the object's size (possibly taking into account object expansion).
- If an object does not fit into the current destination region, then we switch to the next destination region. We may leave a wasted gap at the end of a destination region, which will later be filled with a dummy object. We append the current destination region to the end of the worker's compaction list. We pop the head of the destination-region-list and make that the new destination region.
- If an object also doesn't fit into the next region, we consider it a 'humongous object' which cannot be compacted into a single region and which would span multiple regions. We mark it by setting the 'humongous-start' field in the region to the start address of that object. This field will later be used to skip regular compaction of humongous objects, and treat them specially. During the normal forwarding phase, we don't forward humongous objects, and we don't forward other objects into humongous objects. In other word, we try to not move humongous objects and 'slide other objects around' humongous objects.
- When the phase is finished, we append the remaining destination-list to the end of the compaction-list. The resulting list are all regions that the worker has processed, and serves as the work-list for the compaction phase.
- Finally, we need to process humongous objects. In a serial phase, we look at all regions, bottom to top. If a region has a humongous-start field set, then:
    - If the humongous object starts below the new-top of the space, then it is 'embedded' by other objects around it, and we leave it where it is.
    - Otherwise, the humongous object would leave a gap after the new-top of the space. Slide it down to new-top and bump new-top to the new-end of the humongous object
    - Note that this serial forwarding implies that we also have to serially compact humongous objects. This may prove to be a performance bottleneck. We may try to distribute humongous object copying across worker threads if we can prove that copies of 2 different objects are disjoint (i.e. they don't override each others original locations during copying).

Adjusting Pointers Phase

The Adjusting Pointers Phase is not fundamentally different than before. The main difference lies in how work is divided between the workers. Each worker processes the objects in the regions on its work-list, and updates all reference fields in each object to point to its new location. When processing a region that has been shortened for a humongous object, then the worker must still process the humongous object in the cut-off tail.

Compaction Phase

Similarily, we can also 1:1 adapte G1/Shenandoah's compaction phase.

- Each worker processes its compaction list sequentially.
- It scans the live objects in each region in the compaction list.
- For every live object, find the forwarding address that has been computed in the Forwarding Phase, and copy the object to that address.
- Fill the gap at the end of each destination region with a filler object.