Parallel Garbage Collection

Xiao-Feng Li

Shanghai Many-core Workshop
2008-3-28
- Quick overview on Garbage Collection
- Parallelization topics
  - Traversal of object connection graph
  - Order of object copying
  - Phases of heap compaction
  - Marking of live objects
- Other GC threading topics
- Apache Harmony and GCs
References (Incomplete)

- SIEGWART David, HIRZEL Martin, Improving locality with parallel hierarchical copying GC, ISMM2006
- Ming Wu and Xiao-Feng Li, Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations, IPDPS2007
- Chunrong Lai, Volosyuk Ivan, and Xiao-Feng Li, Behavior Characterization and Performance Study on Compacting Garbage Collectors with Apache Harmony, CAECW-10
- Xianglong Huang, Stephen M Blackburn, Kathryn S McKinley, J Elliot B Moss, Zhenlin Wang, Perry Cheng. The Garbage Collection Advantage: Improving Program Locality, OOPSLA2004
- Hans-J. Boehm, Destructors, finalizers, and synchronization, POPL2003
Agenda

• Quick overview on Garbage Collection
• Parallelization topics
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
• Other GC threading topics
• Apache Harmony and GCs
Garbage Collection: Why?

- GC is universally available in modern programming systems
  - Automatic memory management
  - Largely improves SW develop. productivity
  - Java, C#, Javascript, Ruby, etc.

- GC helps to attack memory wall & power issues
  - In the many-core era
Garbage Collection: What?

• Runtime system identifies dead objects automatically

• Known non-manual approaches
  – Runtime reference counting
    • Runtime overhead, cyclic references
  – Compiler live-range analysis
    • Limited capability
  – Runtime reachability approximation
    • This talk focus
Traverse object connection graph from application’s thread context

Root references in:
- Stacks
- Registers
- Global variables
Garbage Collection: Algorithms

- **Mark-sweep**
  - Trace & mark live objects, sweep dead ones
  - Non-moving

- **Copy**
  - Trace & copy live objects to free area
  - Require free area as copy destination

- **Compact**
  - Mark live objects, compact them together
  - In-place defragmentation
Key Operations in GC

- **Mark-sweep**
  - Trace & mark live objects, sweep dead ones
  - Non-moving
- **Copy**
  - Trace & copy live objects to free area
  - Require free area as copy destination
- **Compact**
  - Mark live objects, compact them together
  - In-place defragmentation
Parallelization Topics

• Next
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
GCs We Developed

- In Apache Harmony
  - GCv4.1
    - Stop-the-world
  - GCv5
    - Parallel STW
  - Tick
    - Mostly concurrent
  - Tick
    - On-the-fly

![Collection Cycle Diagram]
Agenda

• Quick overview on Garbage Collection
• Parallelization topics
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
• Other GC threading topics
• Apache Harmony and GCs
Traversal of Object Graph

• Visit all the nodes in the graph
  – Graph shape is arbitrary
  – Task (mark an object) granularity is small
  – **Question**: load balance among collectors

• Techniques
  – Pool sharing: share a common pool of tasks
  – Work stealing: steal tasks from other collector
  – Task pushing: push tasks to other collector
1. Shared Pool for task sharing
2. One reference is a task
3. Collector grabs task block from pool
4. Pop one task from task block, push into mark stack
5. Scan object in mark stack in DFS order
6. If stack is full, grow into another mark stack, put the full one into pool
7. If stack is empty, take another task from task block

- Block size and stack depth impact
- Need synchronization for pool access
1. Each collector has a thread-local mark-stack, which initially has assigned root set references.
2. Collectors operate locally on its stack without synchronization.
3. If stack is empty, collector steals a task from other collector’s stack’s bottom.
4. If stack has only one entry left, the collector need synchronization access.
5. If stack is full, it links the objects into its class structure (should never happen in reality).

- Stack requires special handling when full
- Need synchronization for task stealings
1. Each collector has a thread local mark stack for local operations.

2. Each collector has a list of output task queues, one for each other collector.

3. When a new task is pushed into stack, the collector checks if any task queue has vacancies. If yes, drip a task from mark stack and enqueue it to task queue.

4. When mark stack is empty, the collector checks if there are any entries in its input task queues. If yes, dequeue a task.

- Task queue mostly is a variable
- No synchronization instruction!!
Agenda

• Quick overview on Garbage Collection
• Parallelization topics
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
• Other GC threading topics
• Apache Harmony and GCs
Order of Object Copying

• Object order largely impacts locality
  – Folks-belief: Allocation order has best locality
    • Maintain allocation order requires compacting
  – Question: copy order for locality

• Techniques
  – Breadth-first copy
  – Depth-first copy
  – Hierarchical order
  – Adaptive object reorder
Breadth First Order

• Cheney’s copying GC

- Pros
  - No additional queue structure

- Cons
  - Probably the worst locality
Depth First Order

- Can be easily achieved with an additional mark stack
  - Stack size proportional to the deepest path

- Pros
  - Better locality

- Cons
  - Stack overhead
    * Deutsch-Schorr-Waite algorithm eliminates stack
Hierarchical Order

- Try to put the connected objects together
  - Limit the queue length of breath-first
  - Or limit the stack depth of depth-first

- Benefits depend on application behavior
Adaptive Object Reorder

• Order the objects according to VM’s heuristics, e.g.,
  – Locality may relate to call graph
  – “Age locality”: objects of same age tend to be accessed closely

• Pros
  – Leverage advantages of runtime

• Cons
  – Runtime overhead may cancel the benefit
Agenda

• Quick overview on Garbage Collection
• Parallelization topics
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
• Other GC threading topics
• Apache Harmony and GCs
Phases of Compaction

- Mapping Collector: 1.5 phases
- Compressors: 2.5 phases
- IBM’s Compactor: 3 phases
- Parallel LISP2 Compactor: 4 phases
- Techniques
  - Leave a single contiguous free space
  - Reserve original object order
  - To squeeze free areas out of the heap
Parallel LISP2 Compactor

1. Live object marking

2. Object repointing

3. Reference fixing

4. Object moving

- Remember target address in object header
• Remember target address in offset table
  – Hence moving does not overwrite target info
• Still, target address remembered in offset table
  – Now it is computes based on mark table
  – No touch of the object heap proper
• Check mark-bit table for free pages
  – Leverage OS virtual memory support to unmap
  – Moving is unnecessary since all are in virtual address space
Agenda

• Quick overview on Garbage Collection
• Parallelization topics
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
• Other GC threading topics
• Apache Harmony and GCs
Marking of Live Objects

• Set a flag to indicate object aliveness
  – Multiple collectors may contend setting
    • Atomic operation might be used
  – Question: synchronization overhead

• Techniques
  – Mark-bit table
  – Mark-byte table
  – Object header marking
  – Section marking
Mark-bit Table

• A separate table for marking status
  – One bit in table for one word in heap
    • Word is the object alignment unit
  – 1/wordwidth of heap used for the table

• Pros
  – Small space overhead
  – Used together with other metadata

• Cons
  – Need atomic operation for bit manipulation
Mark-byte Table

• The same as Mark-bit table, but
  – One byte for one object alignment unit
  – E.g., 1/16 of heap used for the table
    • If object aligned at 16-byte boundary

• Pros
  – Atomic operation not needed
    • When byte is the minimal memory store unit

• Cons
  – Space overhead is higher
Object Header Marking

- Set the marking flag in object header
  - Usually there is a word for meta-info
- Pros
  - No atomic operation needed
- Cons
  - Iterate heap in order to find live objects
    - Slower than mark table scanning
- Marking flag design
  - Single bit or flipping bits
Section Marking

• Mark a section when an object in it is live
  – A section can have multiple objects
  – One flag is used for them, all live or dead together

• Pros
  – Combination of mark table and object marking
    • Small space overhead and no atomic operation

• Cons
  – Floating garbage and live object identification
Agenda

• Quick overview on Garbage Collection
• Parallelization topics
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
• Other GC threading topics
• Apache Harmony and GCs
Other GC Threading Topics

- Thread local objects
- Finalizer processing
- Concurrent collection
- GC and transactional memory
Thread Local Object

• When object is identified as thread-local
  – Synchronization can be eliminated
  – Stack allocation
  – Scalar replacement, object inlining

• Techniques
  – Static escape analysis
  – Dynamic escape analysis (escape detection)
    • Write barrier or read barrier approach
  – Lock reservation, lazy lock
Thread Local Identification

- Static escape analysis
  - Identify life-time TLO
- Dynamic escape analysis
  - Allocation-time TLO, monitor then after
- Lock reservation
  - Not thread local, but lock local
- Lazy lock
  - Allocation-time lock local
Finalizer Processing

• Finalizer is a method
  – Invoked when an object is to be reclaimed
  – Different from C++ destructor

• Finalizers executed in separate thread(s)
  – Finalizable objects are not reclaimed yet
    • Occupy the space; finalizer may create objects
    • And mutators may keep creating new finalizables

• Question: Balance of mutators and finalizing threads
Mutator-Blocking Finalization

- When there are too many finalizers
  - Either start more finalizing threads to compete with mutators for computing resource
    - Preferred when there are idle cores
  - Or suspend guilty mutators until finalizers number drops below a threshold
    - Preferred when cores are all busy
Concurrent collection

• Collecting garbage while application runs
  – For low-pause time (or pauseless)

• Collect with separate collector threads
  – Utilize idle cores
  – Normally single thread is adequate
    • To use least computing resource
    • But if collect slowly, mutators wait for free space

• **Question**: Balance of collection rate and allocation rate
Collectors Work-on-Demand

- Start more collectors when needed adaptively
## GC and Transactional Memory

<table>
<thead>
<tr>
<th>memory management</th>
<th>concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>correctness</td>
<td>dangling pointers</td>
</tr>
<tr>
<td>performance</td>
<td>space exhaustion</td>
</tr>
<tr>
<td>automation</td>
<td>garbage collection</td>
</tr>
<tr>
<td>new objects</td>
<td>nursery data</td>
</tr>
</tbody>
</table>

*Transactional memory is to shared-memory concurrency as garbage collection is to memory management*

Concurrent Copying GC on TM

Lost update problem

- Phil McGachey, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Bratin Saha and Tatiana Shpeisman,
  Concurrent GC Leveraging Transactional Memory, PPoPP'08
Agenda

• Quick overview on Garbage Collection
• Parallelization topics
  – Traversal of object connection graph
  – Order of object copying
  – Phases of heap compaction
  – Marking of live object
• Other GC threading topics
• Apache Harmony and GCs
Apache Harmony

• Primary goal: full JavaSE implementation
  – Class library, competitive VMs, JDK toolset
  – Founded in Apache Incubator in May 2005
  – Became Apache project in Oct 2006

• Facts today
  – 27 committers, 30 commits weekly (currently)
  – 250 messages weekly in mailing list
  – 150 downloads weekly
Harmony DRLVM

• The current default VM of Harmony
• Components
  – Two JIT compilers: fast and optimizing
  – Several GCs: parallel/concurrent
  – Other features: JVMTI, etc.
• Targets
  – Robustness, performance, and flexibility
  – Server and desktop
  – Product-ready
DRLVM Modularity Principles

• Modularity
  – Well-defined modules and interfaces.

• Pluggability
  – Module implementations replicable

• Consistency
  – Interfaces are consistent across platforms.

• Performance
  – Modularity without scarifying performance
Harmony GC Implementations

• GC algorithms
  – Copy: semi-space, partial-forward
  – Compact: sliding-compact, moving-compact
  – Mark-sweep
• And their variants
  – Generational and non-generational
  – Parallel stop-the-world and concurrent
• Modular design enables GC research
Summary

• GC is becoming universal in modern programming systems
  – Important component for many-core
• Parallelization and threading issues in GC
  – Load balance, locality, atomic operation overhead, concurrency, etc.
• Just starting…
  – JIT assistance, OS interaction, HW supports, programming model changes, power, etc.
Thanks! And Questions?

http://harmony.apache.org