Karlsruher Institut für Technologie

Concurrent Garbage Collector

Sprachen für Parallelprogrammierung: Seminar 6
Emre Kasif Selengin

IPD Snelting, Lehrstuhl Programmierparadigmen

```
package garbageCollectionTests;

public class Programm {

    public static void main(String[] args) {

        Object iObject = new Integer(0);

        for (int i = 0; i <= Integer.MAX_VALUE; i++) {
            iObject = new Integer(i);
        }
```

Übersicht

Karlsruher Institut für Technologie

1. Was ist Garbage Collection
2. Terminologie
3. Algorithmen
   - GC
     Erreichbarkeit
   - Generational GC
   - Parallel GCs
   - Concurrent GC
4. Die Heutigen GCs in Programmierumgebungen
5. Fazit

Was ist Garbage Collection (GC)

- Die nicht-nativen und managed Sprachen werden beliebter. Speicherallokation wird automatisch.

```
Object o = new SomeComplexObject();
// ... work with o ...
o = null;
```

- Hier wird der Konstruktor der Klasse SomeComplexObject gerufen und der Speicher wird automatisch das Objekt zugeteilt.
- Wenn es später nicht mehr gebraucht wird, muss es gelöscht werden. Da kommt der GC ins Spiel.

Was ist Garbage Collection (GC)

- GC muss erkennen, welche Objekte nicht mehr gebraucht sind. Tote Objekte können dann gelöscht werden.
- Er muss sicher sein, dass tote Objekte wirklich tot sind.
- Er muss es auch schnell machen.

Was ist Garbage Collection (GC)

- Der Konzept stammt fünfziger Jahren. Zuerst für die Sprache LISP benutzt.
- Intensiv geforscht wird aber nur in letzten 15 Jahren. Effizienz war für Praxis nicht genug. Man hat lieber explizite Speicherverwaltung benutzt.

Was ist Garbage Collection (GC)

- Naive Implementation der GC muss das gesamte Programm anhalten, damit er die Objekte durchlaufen kann: Stop-The-World
- Overhead war zu groß.
- Diese führt zu zwei große Probleme:
  - Programme werden langsamer: Durchsatz (Throughput)
  - GC fügt lange Pausen zu: Wartezeit (Latency)
- Forscher (auch in Firmen wie Microsoft und Sun) vorschlagen neue Ideen für diese Probleme. Parallelisierte GCs gehören dazu.

Übersicht

Karlsruher Institut für Technologie

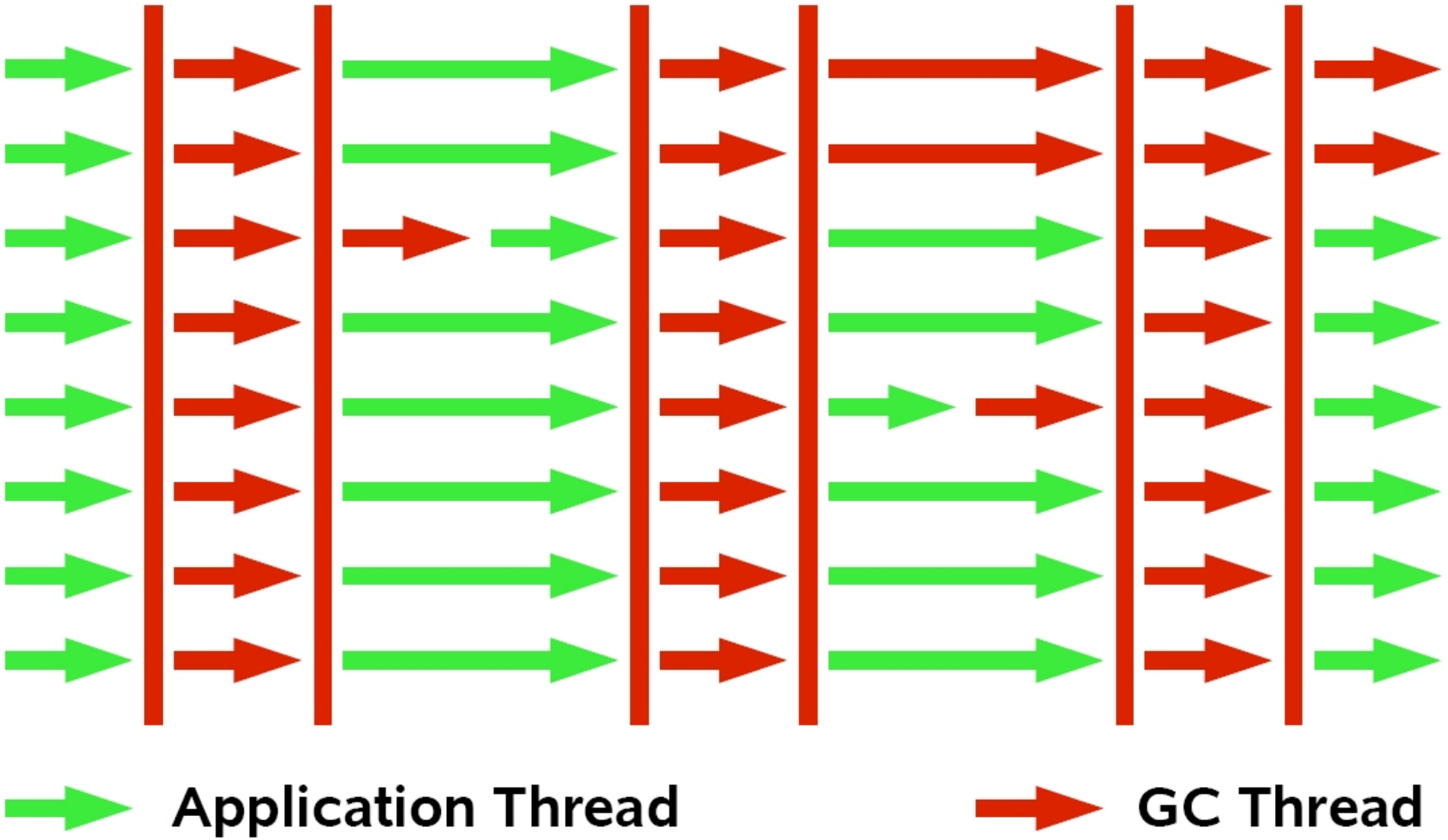
1. Was ist Garbage Collection
2. Terminologie
3. Algorithmen
   - GC
     Erreichbarkeit
   - Generational GC
   - Parallel GCs
   - Concurrent GC
4. Die Heutigen GCs in Programmierumgebungen
5. Fazit

Terminologie: Concurrent vs. Parallel

Karlsruher Institut für Technologie

- Diese Wörter haben ähnliche Bedeutungen. Manchmal als Synonym benutzt.
- Bei der GC-Forschung macht es doch ein Unterschied.

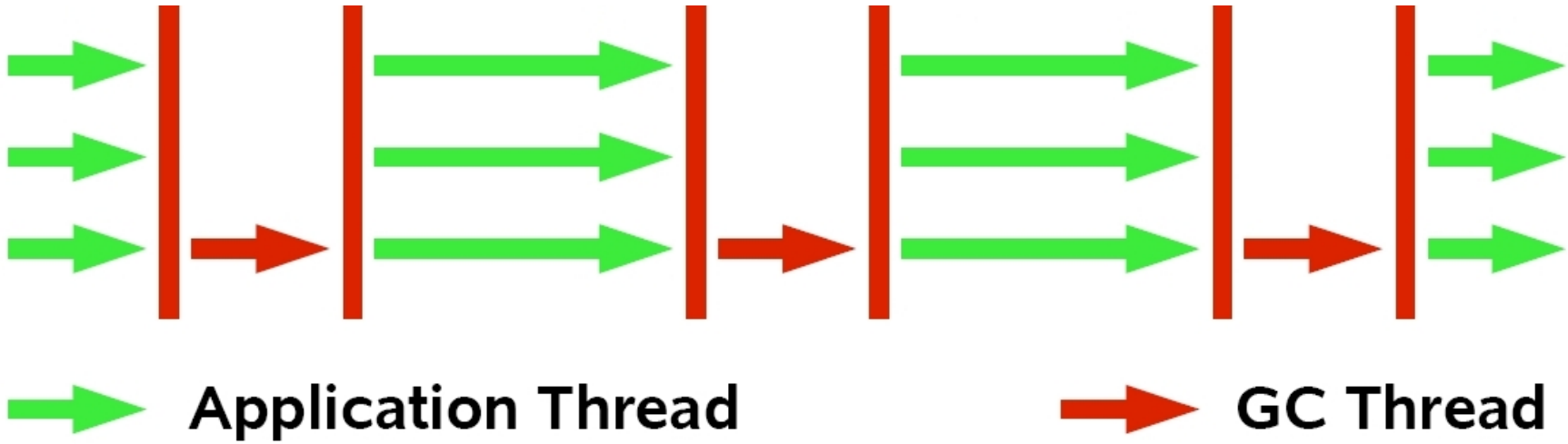
Terminologie: Concurrent vs. Parallel



[1]

[1] Grafik aus [13]

GC ohne Parallelismus

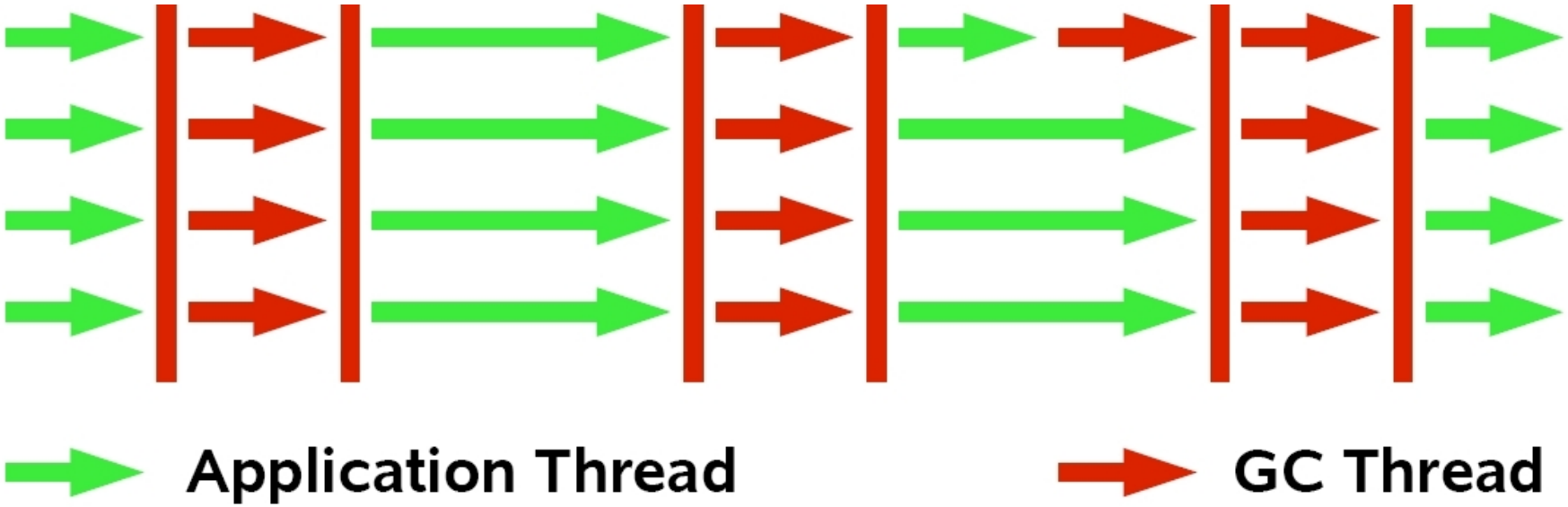


[2]

[2] Grafik aus [13]

Terminologie: Parallel GC

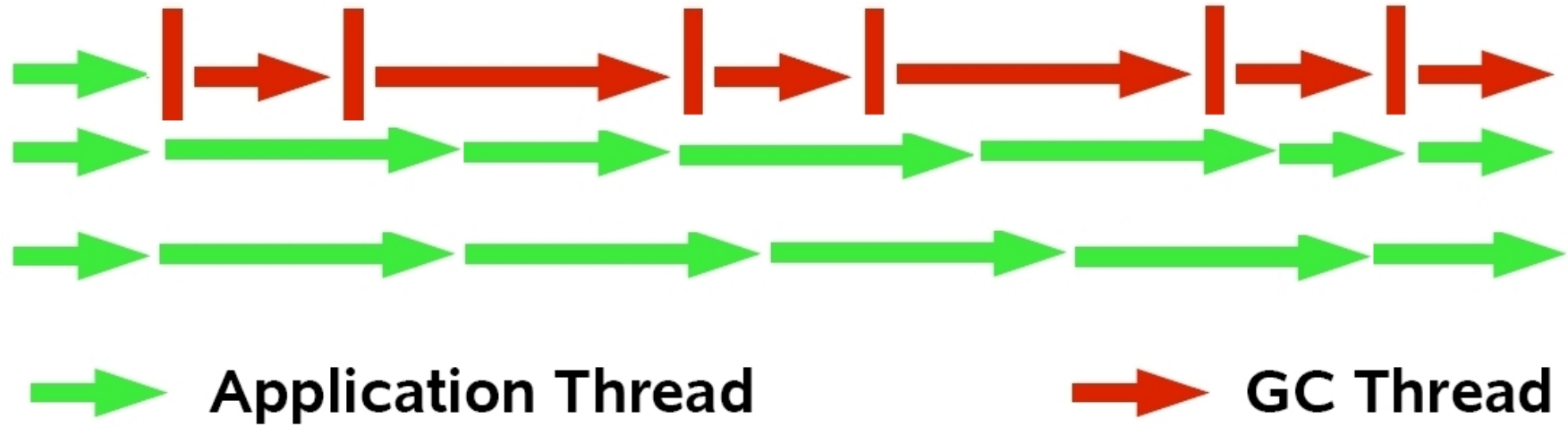
- Die Arbeit der GC bleibt gleich aber sie wird zwischen den Prozessoren (oder Threads) verteilt.



[3] Grafik aus [13]

Terminologie: Concurrent GC

- Concurrent = Nebenläufig.
- Die eigentliche Anwendung läuft ungestört. GC lauft neben den Threads des Programms.



[4] Grafik aus [13]

Terminologie: Concurrent GC

Karlsruher Institut für Technologie

- Vollständige Nebenläufigkeit ist Praktisch unmöglich.
- GCs müssen auch wissen, es eine Änderung gibt. Nachrichten müssen geschickt werden.
- Write Barrier in JVM: Nach jedes Update kleine Operation wird gerufen.
- Zwei bis fünf Instruktionen für jede Schreibzugriff.
- Write Barrier gibt es schon in JVM, also weniger Problem. Auch normale GCs brauchen dieses Konzept.

Übersicht

Karlsruher Institut für Technologie

1. Was ist Garbage Collection
2. Terminologie
3. Algorithmen
   - GC
     Erreichbarkeit
   - Generational GC
   - Parallel GCs
   - Concurrent GC
4. Die Heutigen GCs in Programmierumgebungen
5. Fazit

Naive Garbage Collector

Karlsruher Institut für Technologie

- Garbage Collectors sind praktisch nicht deterministisch. Wir können nicht vorhersagen, dann sie laufen.
- Wenn der Grenze der Speicherbenutzung überschritten ist, fängt GC an. Danach arbeitet er zyklisch.
- Man kann in modernen Systemen (.Net oder Java) auch GC zur Arbeit zwingen (Java: System.gc()).
- *Stop-the-world-Pausen* sind sehr Problematisch.

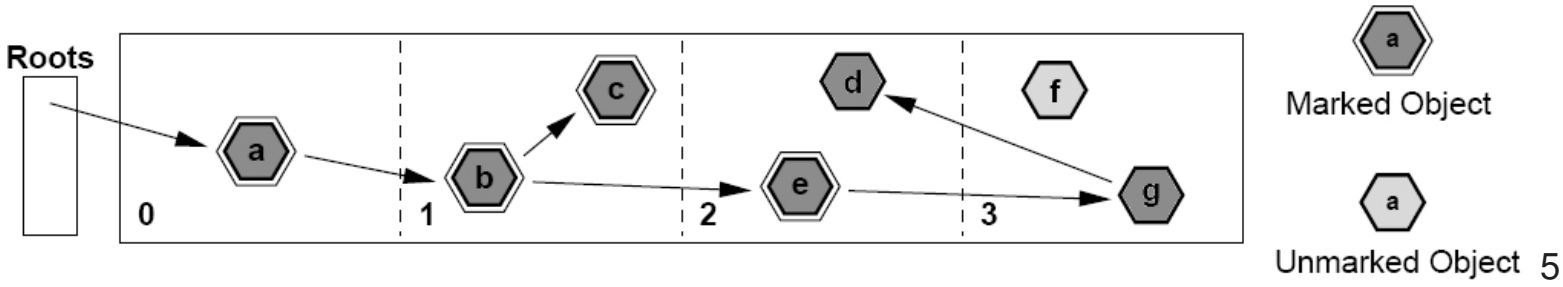
Naive Garbe Collector - Erreichbarkeit

- Die Erreichbarkeit ist durch die Referenzen getestet. Wenn eine Methode noch Referenz zu einem Objekt hat, ist es erreichbar.
- Referenzen können von den Programmen gezählt werden (Reference Counting). Wenn ich ein Objekt brauche melde ich mich an. Danach melde ich mich ab.
- Ist die Referenzanzahl auf einem Objekt 0, ist das Objekt tot.
- Diese Methode wird nicht mehr benutzt (viele *Nachteile*).

Erreichbarkeit: mark-and-sweep

- Anstatt *Reference Counting* geht ein GC durch den Speicher.
- Er markiert die Objekte, die durch anderen erreichbar sind. Danach wird gelöscht (*sweep*).

  Roots (Globale Datenbereiche, Registers, Stack) sind gute Anfangspunkte.



- Frage: Was, wenn das Programm ein Referenz ändert?

[5] Grafik aus [1]

Erreichbarkeit: mark-and-sweep

Karlsruher Institut für Technologie

- Antwort: Ein Programm kann nichts ändern. Graph muss stabil sein.
- Markierungsalgorithmus wird weiter benutzt. Löschalgorithmus wird mit Weiteren verbessert.
- Mark-And-Sweep braucht Stop-the-world-Verfahren. Concurrent GC ist nicht möglich.
- Markierungsalgorithmus muss weiterentwickelt werden.

Erreichbarkeit: Tri-Color Marking

- Eine Variante von Mark-and-sweep: Statt 2 Zustände zu haben, haben die Objekte drei Farben.
  - Schwarz: Sicherlich wird benutzt
  - Grau: wird benutzt und sein Unterobjekte sind noch nicht kontrolliert
  - Weiß: wurde noch nicht kontrolliert. Wenn der Graph gar kein Grauen Objekte hat, die Weißen werden sicherlich nicht benutzt und können gelöscht werden.
- Schwarze Objekte haben niemals direkten Kontakt zu Weißen
  - Anfang: Nur Objekte, die aus Roots erreichbar sind, werden grau markiert.
  - Die Unterobjekte von Grauen werden kontrolliert. Sie werden auch Grau markiert.
  - Sind alle Unterobjekte eines grauen Objekts kontrolliert, wird es Schwarz markiert.
  - Wenn alle Grauen weg sind, ist der Graph nicht stabil.
- Nebenläufigkeit wird mit verschiedenen Methoden erreicht.

Generational GC

- Hat mit der Parallelität nicht zu tun aber wichtig, weil moderne GCs funktionieren nach dieses Konzept.
- Empirische Beobachtungen sagen, dass die neue Objekte früher sterben und alte Objekte selten Referenzen auf die Neuen haben (old-to-young).
- Wir können Objekte nach “Alter” trennen. Dadurch werden Objekte mit gleichen Eigenschaften zusammengefasst.
- In .NET gibt es drei, in JVM zwei [6] Generationen.
- Diese Klassen (Generationen) der Objekten haben daher mit verschiedenen Techniken bearbeiten. Beispiele kommen gleich.

[6] und noch eine für dazwischen

Parallel GCs

- Man nennt diese auch throughput-Collectors. Die verbessern den Durchsatz. Performancegewinn durch Parallelismus.
- GCs können parallelisiert werden, indem man die Datenmenge der normalen GC (Heap) zu Threads verteilt.
- Mark-and-sweep zu parallelisieren wurde auch vorgeschlagen (wurde nicht benutzt in Praxis)
- Parallel Collector in JVM ist eine Stop-the-World-Algorithmus. Alle CPUs werden ausgenutzt um die Pause kurz zu halten.
- Ist die Algorithmus zu komplex, muss man auch Synchronisation einbauen.

Parallel GCs

- Der alte Version der PGC in JVM läuft nur über die Young-Objects. Der neuere Version, Compacting PGC läuft über alle Generationen.
- Wenn eine Generation voll ist, da läuft der GC. Wenn beide voll sind, dann läuft der GC auf beiden.
- PGC benutzen mark-and-sweep ähnlichen Algorithmen, die für seriellen GCs gedacht sind:
- Der alte PGC benutzt mark-and-copy (in JVM)
- Der neuere PGC benutzt eine Variante der Mark-and-Compact (Daher hat den Namen Parallel Compacting Collector)

Concurrent GC

- Diese Versuchen die Wartezeit zu verkürzen (deshalb low-latency), indem man ein GC-Prozess neben den Anwendung gleichzeitig arbeiten lässt.
- Es wurde [5] vorgeschlagen, dass man dafür Tri-Colour markierung benutzt.
- Darauf basiert sich die heutigen Concurrent Mark-Sweep in JVM[1].
  - Sie braucht write-barriers, um sich zu informieren, wo (in welchen Region) es während der Markierung etwas geändert wurde.
  - Sie braucht auch Speicherplatz für Objekte.
  - Zwei Markierungsphasen: Eine läuft komplett nebenläufig. Danach muss er die Anwendung anhalten, um die Änderungen zu kontrollieren.
  - In remark Phase kann Parallelität(wie in oben) vorkommen. Deshalb ein Hybrid GC.

Concurrent Mark-Sweep in JVM

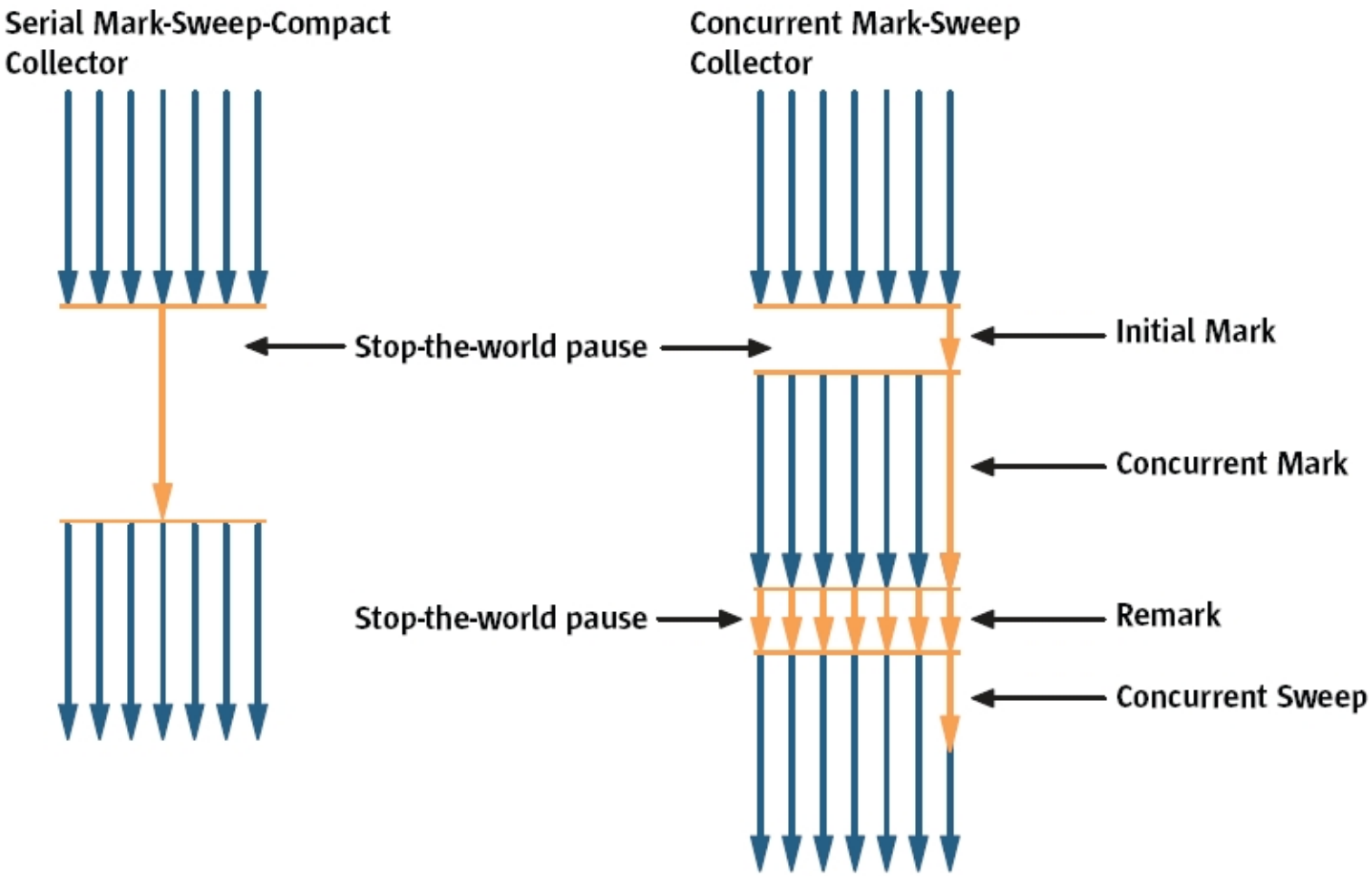


Figure 7. Comparison between serial and CMS old generation collection [7]

[7] Grafik aus [3]

Concurrent Mark-Sweep in JVM

Karlsruher Institut für Technologie

- verkürzte Wartezeiten für GC in Old-Generation.
- Aber:
  - Wartezeiten in Young-Generation-GC
  - extra Speicherverbrauch
  - Reduzierung in Durchsatz
  - Deshalb ist er nicht immer beliebt.

Übersicht

Karlsruher Institut für Technologie

1. Was ist Garbage Collection
2. Terminologie
3. Algorithmen
   - GC
     Erreichbarkeit
   - Generational GC
   - Parallel GCs
   - Concurrent GC
4. Die Heutigen GCs in Programmierumgebungen
5. Fazit

Die Heutigen GCs

Karlsruher Institut für Technologie

Microsoft-Welt: .Net

- Bis .Net 4.0 gab es keine Parallele oder Nebenläufige GCs.
- Nur Generationale GCs: mark-and-sweep auf Gen0 und Gen1. mark-and-compact auf 2.
- Weitere Optimierungen sollen existieren
- Aber jetzt auch ein CGC für Old-Generation (Hat aber die ähnliche Probleme wie in JVM)

Java-Welt

- JVM bietet einen seriellen und drei GCs mit Paralleismus.
- PGCs für verschiedene Generationen und CGC für Old-Generation.
- CGC kann durch Overhead problematisch werden.
- Es wird aber weiter geforscht: Eine neue CGC namens Garbage First (G1)GC. Der fertige Version wird mit Java7 veröffentlicht.

Praxis: Wann ist es sinnvoll parallelisierte CGs zu benutzen?

- PGC können benutzt werden, wo es mindestens zwei Prozessoren gibt.
- Sogar in Echtzeitsystemen versucht man diese zu benutzen aber lange Pausen sind Problematisch.
- CGC können benutzt werden um die Wartezeiten zu verkürzen, wenn man den GC einen Thread schenken kann.
- Alle moderne GCs können parametrisiert und zu einem Programm angepasst werden.

Fazit

- Teilalgorithmen sind wie Entwurfsmuster für GC.
- Vielleicht ist die ultimative Lösung ist noch nicht gefunden aber es gibt jetzt mehr Gründe, automatische Speicherverwaltung zu wählen.
- Mit Parallelismus sind GCs noch besser worden und die Geschichte geht weiter.
- PGC sind jetzt sicher aber CGC haben auch Chancen.

Fragen? [8]

[8] Referenzen sind in meiner Ausarbeitung.