The free lunch is over

Java 7: Das Fork-Join-Framework für mehr Performance

Markus Günther, Martin Lehmann

Mehr Performance wird dem Softwareentwickler nicht länger durch immer schnellere Prozessoren geschenkt. Stattdessen muss man mehrere parallele Prozessoren effizient ausnutzen. Wie aber implementiert man parallele Algorithmen fehlerfrei und effizient? Klassische Java-Threads reichen nicht aus, da sie keine geeignete Abstraktion für die inhärente Parallelisierung eines Problems mitbringen. Java 7 bietet mit dem Fork-Join-Framework eine leichtgewichtige Alternative und unterstützt den Entwickler bei der Umsetzung parallelisierbarer Aufgaben. Am Beispiel der Berechnung von Fraktalen zeigt der Artikel, wie ein parallelisierbares Problem in Java durch Zerlegung in Teilprobleme (Fork) und anschließende Zusammenführung der Teilergebnisse (Join) umgesetzt werden kann.

Paradigmenwechsel

„The free lunch is over". Mit diesen schönen Worten hat Herb Sutter in [Sutt05] beschrieben, dass das Zeitalter des Schneller-Höher-Weiter vorbei ist, in dem man sich als Softwareentwickler zurücklehnen und auf immer schnellere Performance durch verbesserte Hardware verlassen konnte. Warum anstrengen, wenn Moore's Law gilt?

Weitere Leistungssteigerungen einzelner CPUs sind nicht mehr im großen Maßstab zu erwarten, bedingt durch Fertigungsgröße und physikalische Grenzen. Die Hardwarehersteller haben einen Paradigmenwechsel hinter sich, wenden nun Moore's Law auf andere Art und Weise an, indem sie Mehrkern-CPUs bauen: Dual-Core, Quad-Core, Octo-Core sind State-of-the-Art. CPUs mit einigen Dutzend bis mehreren Hundert Prozessoren sind absehbar.

Doch die theoretische Performance einer Multicore-Umgebung will ausgenutzt werden. Um die vielen Prozessoren auslasten zu können, müssen sich Anwendungsentwickler auf den Paradigmenwechsel einstellen. Das weitverbreitete Programmiermodell des „Shared Mutable State" für die Anwendungsentwicklung ist nicht mehr überall sinnvoll nutzbar (Details siehe [iX12]).

Das Fork-Join-Framework in Java 7

Auch die Entwickler der Java-Plattform haben dies erkannt und bieten sukzessive Support für die Herausforderung Multicore. Die „Concurrent Utilities" [JSR166] bringen schon mit Java 5 *„medium-level utilities that provide functionality commonly needed in concurrent programs"*.

Java 7 bringt nun das Fork-Join-Framework mit, das im JSR 166y spezifiziert ist. Damit lassen sich Probleme lösen, die einfach zu parallelisieren sind. Unter Parallelität versteht man die Ausführung verschiedener disjunkter Aufgaben zur selben Zeit. Sie beeinflussen sich gegenseitig nicht und sind nicht von den Resultaten anderer Aufgaben abhängig. Man spricht auch von „embarrassingly parallel" oder „leichter Parallelisierbarkeit", wenn sich komplexe Probleme besonders einfach in Teilprobleme zerlegen lassen. Beispiele sind Algorithmische Optimierungsverfahren wie Evolutionäre Algorithmen, Such-Heuristiken (wie Lokale Suche, Simuliertes Abkühlen), Rendering-Algorithmen, Berechnung von Fraktalen u.v.m.

Teile und herrsche

Diese Problemklasse basiert auf Teile-und-Herrsche und zerlegt durch Rekursion ein großes Problem in kleinere, unabhängig voneinander lösbare Teilprobleme. Lösungen der Teilprobleme werden zum Gesamtergebnis zusammengeführt:

```
loeseProblemMitForkJoin (Problem p)
  if (p ist klein genug)
    loese p direkt und sequenziell
  else
    teile p in unabhängige Teilprobleme p1 und p2      // Split
    loese p1 und p2 unabhaengig voneinander (rekursiv) // Fork
    fuehre Teilergebnisse von p1 und p2 zusammen       // Join
```

Nachstehend werden wir nur noch auf Fork und Join eingehen. Die Komplexität von Split in Teilprobleme darf man jedoch nicht unterschätzen: Bei trivialen Beispielen ist der Split über die Zieldatenstruktur einfach möglich (Beispiel unten: Mandelbrot-Berechnung, Split durch die Halbierung des Berechnungsintervalls). Man muss den Split-Faktor („Wie teile ich den Workload auf? Immer uniform?") und den Branching-Faktor („In wie viele Anteile breche ich den Workload auf?") geeignet wählen, um gute Ergebnisse bzgl. der Parallelisierung zu erreichen. Und das ist für manche Problemdomänen nicht trivial. Nicht auf jeden Fork muss ein Join erfolgen: Manche Probleme lassen sich so aufteilen, dass die Teilprobleme auf einer gemeinsamen Zieldatenstruktur parallel arbeiten können, sofern deren Datenbereiche voneinander unabhängig sind. In diesem Fall ist eine abschließende Ergebniskombination im Join nicht nötig, da die Lösungsfindung direkt auf der Datenstruktur erfolgt. Die Datenstruktur *ist* also die Lösung. Unser nachstehendes Beispiel der Mandelbrot-Berechnung fällt in diese Kategorie.

Das Fork-Join-Framework unterstützt bei der Zerlegung in Teilprobleme und der Rekombination der Ergebnisse. Eine Task T_1 wird rekursiv in Teilprobleme zerlegt, die Teilergebnisse werden zusammengeführt (s. Abb. 1). Dieser Vorgang wiederholt sich solange, bis die Teilprobleme klein genug sind, um sie sequenziell zu lösen (vgl. Ebene n=k mit 2^k Teilproblemen). Ein übergeordnetes Problem erhält die Lösung seiner Teilprob-

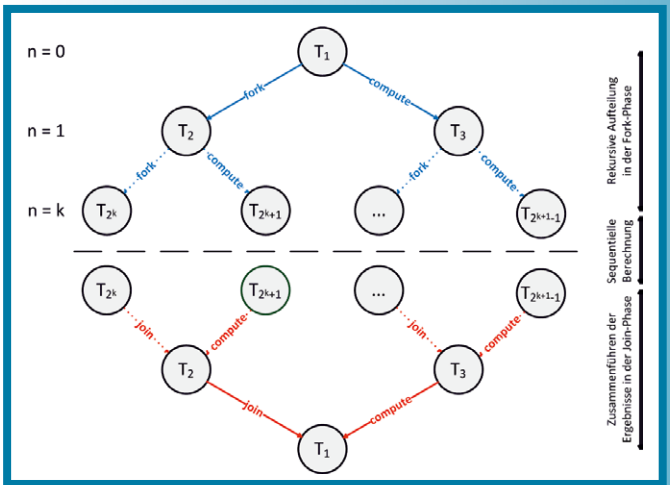

Abb. 1: Fork-Join-Baum

leme und fügt diese in der Join-Phase zusammen. Dies wiederholt sich solange, bis das Ergebnis von T_1 vorliegt.

Klären wir zunächst die Frage, warum man mit Java-Bordmitteln den oben skizzierten Algorithmus nicht sinnvoll abbilden kann und warum man dafür überhaupt ein neues Java-Framework benötigt. Naiv könnte man jede Task als eigenen Java-Thread implementieren und Teile-und-Herrsche mit den Thread-Funktionen `start` und `join` implementieren. Das funktioniert prinzipiell, ist aber aus Effizienzgründen nicht praktikabel (siehe [Goetz06], Kapitel 6):

- Große Probleme sind so nicht lösbar, da zu viele Tasks und damit Threads erzeugt würden, was schnell die Systemgrenzen sprengt.
- Erzeugung und Beendigung von Threads hat zu viel Overhead hinsichtlich Laufzeitperformance und Speicherverbrauch.

Offensichtlich ist also der Einsatz eines Thread-Pools für solche Probleme obligatorisch. Und einen solchen für die Problemklasse optimierten Thread-Pool stellt das Fork-Join-Framework bereit.

Framework-Klassen: Thread-Pool, Threads, Tasks

Die Klassenstruktur des Fork-Join-Frameworks ist mit wenigen Klassen sehr übersichtlich, wie Abbildung 2 zeigt. Der `ForkJoinPool` ist ein Thread-Pool, ähnlich des schon länger im JDK befindlichen `ThreadPoolExecutor`, allerdings mit für Fork-Join optimierten und konfigurierbaren Eigenschaften (so ist beispielsweise die Anzahl der Pool-Threads konfigurierbar, z. B. über `java.lang.Runtime#availableProcessors`). Die `ForkJoinWorkerThreadFactory` instantiiert bei Bedarf die `ForkJoinWorkerThread`s.

Ein `ForkJoinWorkerThread` ist ein Thread im Thread-Pool. Eine `ForkJoinTask<V>` definiert eine abzuarbeitende Task, implementiert `Runnable`. Zwei Unterklassen stellen weitere Convenience-Methoden für den Umgang mit Berechnungsergebnissen bereit: `RecursiveAction` für Probleme ohne Rückgabetyp, `RecursiveTask<V>` für Probleme mit Rückgabetyp. Eine `ForkJoinTask<V>` kann an einen beliebigen `ForkJoinWorkerThread` zur Abarbeitung übergeben werden.

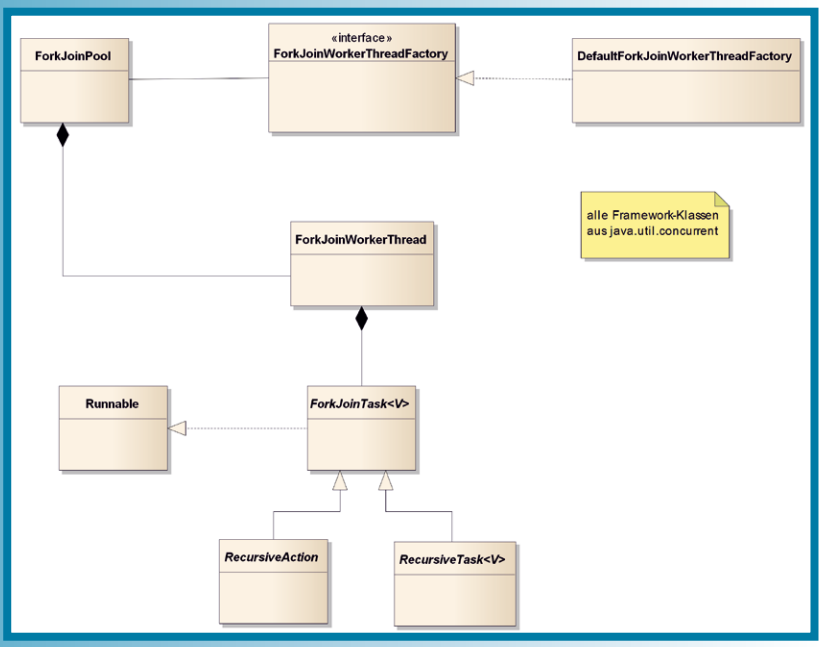

Abb. 2: Klassenstruktur des Fork-Join-Frameworks

Abarbeitung von Tasks durch Threads

Die Abbildung eines Teilproblems erfolgt in einer `ForkJoinTask<V>`. Solch eine `ForkJoinTask<V>` kann an einen beliebigen Worker-Thread zur Abarbeitung übergeben werden.

Jeder Worker-Thread verwaltet seine Tasks in einer „Deque“ (Double-Ended Queue, s. Abb. 3 und [Lea00]).

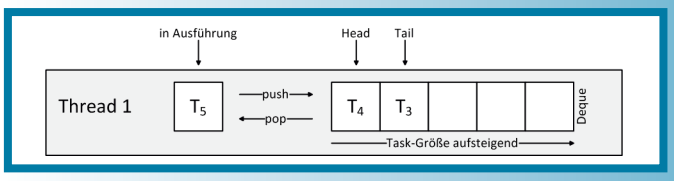

Abb. 3: Jedem Thread ist eine eigene Deque zugeordnet. Für Thread 1 steht die Bearbeitung von T_5 an. T_4 und T_3 warten auf die Bearbeitung

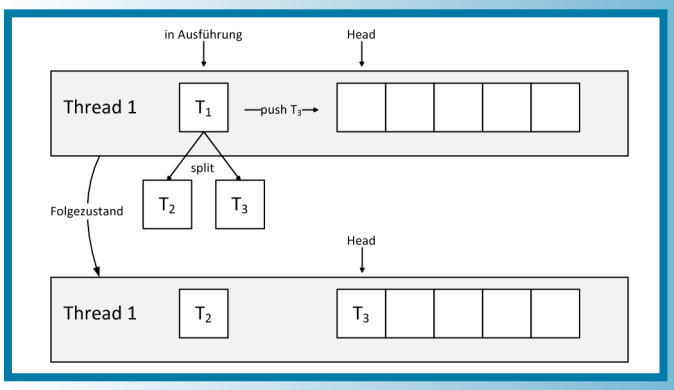

Abb. 4: Verarbeitung von T_1 (oben) und Ergebnis (unten)

Randnotiz: Die Implementierung der Deque im JDK ist hochoptimiert. Intern wird ein Array benutzt und auf die normalen Checks der Array-Grenzen bewusst verzichtet (unter Zuhilfenahme von `sun.misc.Unsafe`). Sämtliche Garantien des Java-Memory-Modells werden ausgereizt.

Wird ein Problem in Teilprobleme aufgeteilt, so werden diese an den Anfang der Deque desjenigen Threads eingestellt, der das Problem zerlegt hat (push). So ist zu einem gewissen Grad sichergestellt, dass der Thread weiterhin beschäftigt bleibt.

Abbildung 4 zeigt den Zustand eines Threads während und nach der Bearbeitung einer Task: T_1 wird in die Teilprobleme T_2 und T_3 zerlegt. Während der Thread T_2 direkt weiter bearbeitet, stellt er das Teilproblem T_3 zurück in die Deque (push).

Work-Stealing durch andere Threads

Leert sich die Deque eines Worker-Threads, so droht er beschäftigungslos zu werden, was ineffizient ist, wenn an anderer Stelle noch Arbeit ansteht. Dem wirkt die Strategie des sogenannten „Work-Stealings“ entgegen: Hat ein Worker-Thread keine Task mehr, so kann er auf die Deque eines anderen Threads zugreifen und sich daraus Arbeit beschaffen (s. Abb. 5). Ist das nicht möglich, so stellt sich der beschäftigungslose Thread mit niedriger Priorität in den Hintergrund.

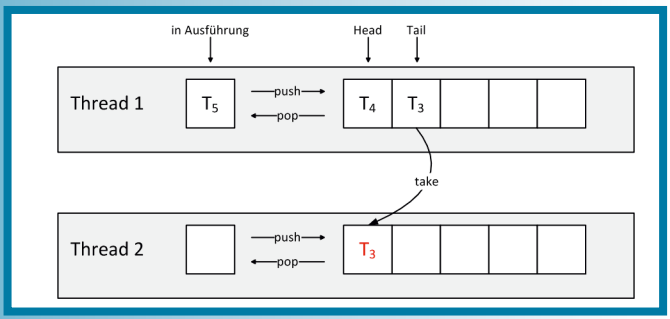

Abb. 5: Work-Stealing: Der beschäftigungslose Thread 2 stiehlt Task T_3 aus der Deque von Thread 1 (take)

Die Eigenschaften der Datenstruktur Deque sind entscheidend, um beim Work-Stealing den Synchronisationsaufwand zwischen Threads minimal zu halten:

- Ein Thread arbeitet nur nach LIFO (last in, first out) auf seiner Deque (push/pop).
- Ein Thread kann nach FIFO auf Deques anderer Threads zugreifen, um eine Task von deren Ende zu stehlen (take).

Dieses Vorgehen garantiert eine effiziente Durchführung:

- Beim Work-Stealing werden die „großen" Tasks übernommen, die es sich zu stehlen lohnt: Das Framework macht sich zunutze, dass Teilprobleme gemessen an ihrem Workload immer kleiner werden. Dadurch, dass Work-Stealing nach FIFO erfolgt, die Tasks aber nach LIFO in die Deques eingelagert werden, holt sich ein stehlender Worker-Thread ein Teilproblem mit hohem Workload.
- In der Regel konkurrieren Besitzer-Thread und Work-Stealer-Thread nicht um Tasks, da der Besitzer-Thread Tasks vom Anfang der Deque entnimmt und der Work-Stealer-Thread von deren Ende. Das minimiert die Thread-Contention.
- Ein Thread merkt sich, welche Threads ihm Arbeit gestohlen haben. Wird er selbst beschäftigungslos, stiehlt er zuerst von diesen zurück (Stealback), anstatt andere, zufällige Aufgaben anzugehen.

Anwendungsbeispiel „Mandelbrot-Menge"

Am Beispiel der Berechnung einer Mandelbrot-Menge (s. Abb. 6) zeigen wir, wie ein typisches parallelisierbares Problem mit Fork-Join in Java umgesetzt werden kann und welche Stellgrößen es gibt, die Einfluss auf die Leistung des Programms haben. Die Berechnung der Mandelbrot-Menge ist gut parallelisierbar, da jeder Funktionswert unabhängig von allen anderen ist:

- Unser Programm berechnet eine Mandelbrot-Menge für eine gegebene Bildauflösung.
- Die Teilergebnisse werden in ein Integer-Array abgelegt. Einzelne Ergebnisse müssen nicht im Join kombiniert werden, da direkt auf der Zielstruktur gearbeitet wird. Unsere Klasse `MandelbrotAction` erbt daher von `RecursiveAction`.

Da alle Worker-Threads Zugriff auf das gesamte Integer-Array haben, müssen wir die Zerlegung in Teilprobleme so vornehmen, dass einzelne Threads disjunkte Bereiche des Arrays bearbeiten. Jedem Teilproblem geben wir zwei Indizes `from` und `to` für die Bereichsgrenzen mit:

```
public class MandelbrotAction extends RecursiveAction {
  private int   maxJobSize;
  private int   from;
  private int   to;
  private int[] imageData;

  public MandelbrotAction(
      int[] imageData,
      int from, int to,
      int maxJobSize) {
    this.imageData = imageData;
    this.from = from;
    this.to = to;
    this.maxJobSize = maxJobSize;
  }
```

Jede Ableitung von `RecursiveAction` oder `RecursiveTask<V>` erfordert die Implementierung von `compute`. Diese Methode ist dafür verantwortlich, das Problem weiter zu zerlegen, oder – falls das Problem schon klein genug ist – es im aktuellen Thread direkt zu lösen:

```
@Override
protected void compute() {
  int size = to - from;

  if (size <= maxJobSize) {
    computeDirectly();  // direkte Berechnung
    // (hier nicht weiter ausgeführt)
  } else {
    parallelize();
  }
}
```

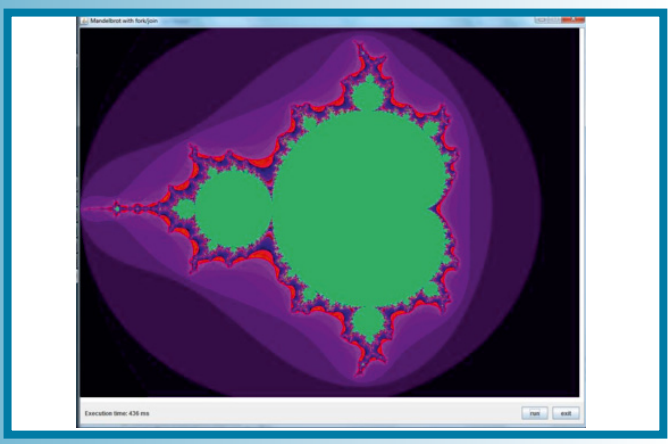

Abb. 6: Screenshot des Mandelbrot-Beispiels

Workload: Nicht zu klein und nicht zu groß!

In unserem Fall berechnet sich der Workload durch die Größe des zu berechnenden Teilbildes. Die Instanzvariable `maxJobSize` dient als Referenzwert und kann bei Aufruf des Konstruktors parametriert werden. `maxJobSize` ist *die* zentrale Stellgröße (neben Thread-Pool-Größe, Branching- und Split-Faktor):

- Wird `maxJobSize` zu klein gewählt, so teilen wir das Problem in unnötig kleine Teilprobleme auf, die schlimmstenfalls dafür sorgen, dass der Threading-Overhead im Verhältnis zur Berechnungszeit pro Teilproblem zu hoch wird. Das verringert den parallelen Durchsatz.
- Wird `maxJobSize` zu groß gewählt, so ist das Problem nicht optimal parallelisiert, es könnte also durch eine weitere Zerlegung profitieren.

Die richtige Wahl von `maxJobSize` ist daher immer der Problemstellung anzupassen. Es ist ratsam, verschiedene Parametrierungen zu erproben und aufeinander abzustimmen.

Zerlegung in Teilprobleme

Die Methode **parallelize** ist für die Zerlegung in Teilprobleme verantwortlich. Unser rekursiver Abstieg hat einen Branching-Faktor von zwei, d. h. wir zerlegen das Problem in zwei Teilprobleme mit je gleichen Workloads (uniformer Split). Mit entsprechenden Bereichsgrenzen erzeugen wir zwei neue Instanzen von **MandelbrotAction** als **left** und **right**:

```
private void parallelize() {
  int half = (to - from) >> 1;

  MandelbrotAction left = new MandelbrotAction(
      imageData, from, from + half);
  MandelbrotAction right = new MandelbrotAction(
      imageData, from + half, to);
  invokeAll(left, right);
}
```

invokeAll ist eine Convenience-Methode des Frameworks, die uns das korrekte Abspalten (Fork) und Zusammenführen (Join) von **left** und **right** abnimmt. Durch den Join innerhalb von **invokeAll** wird überdies sichergestellt, dass Änderungen an unserem Integer-Array dem ausführenden Thread des übergeordneten Problems sichtbar gemacht werden. Synchronisationsprimitive oder **volatile** in der Array-Deklaration sind also nicht nötig. Generell ist die Nutzung von **invokeAll** einem manuell implementierten Fork/Join vorzuziehen: Setzt man Fork/Join manuell falsch um (vgl. dazu [Gross12]), so degeneriert der Berechnungsprozess zu einer sequenziellen Abarbeitung – Parallelisierung ade!

Parametrierung des Mandelbrot-Beispiels

Unsere **MandelbrotAction** testen wir in einem **Testbed**, um eine geeignete Parametrierung zu finden. Unsere Eingabeparameter sind Thread-Pool-Größe sowie der Grenzwert, der festlegt, ob ein Teilproblem weiter zerlegt wird:

```
public class MandelbrotTestbed extends Testbed {
  private int REPETITIONS = 10;
  private long measure(int forkJoinPoolSize, int maximumJobSize,
                       int imageWidth, int imageHeight,
                       int[] imageData) {
    MandelbrotAction proc =
                new MandelbrotAction(imageData, 0, imageHeight,
                maximumJobSize);
    ForkJoinPool forkJoinPool = new ForkJoinPool(forkJoinPoolSize);
    long start = System.nanoTime();
    forkJoinPool.invoke(proc);
    long end   = System.nanoTime();
    return (end-start);
  }
  public void simulate() {
    int[] imageData = new int[2048*2048];
    int[] poolSizes = new int[] { 1, 1, 2, 4, 8, 16, 32, 64, 128 };
    for (int i = 0; i < poolSizes.length; i++) {
      int maximumJobSize = 1;
      while (maximumJobSize <= 2048) {
        long[] times = new long[REPETITIONS];
        for (int j = 0; j < REPETITIONS; j++) {
          times[j]=measure(poolSizes[i], maximumJobSize,
                           2048, 2048, imageData);
          collectGarbage();
        }
        double mean   = mean(times);
        double stddev = stddev(times);
        System.out.println(poolSizes[i] + "\t" + maximumJobSize +
                           "\t" + mean + "\t" + stddev);
        maximumJobSize = maximumJobSize << 1;
      }
      System.out.println();
```

➔

```
    }
  }
  public static void main(String[] args) {
    new MandelbrotTestbed().simulate();
  }
}
```

Leistungsmessung

Das Testbed führt einen Mikrobenchmark durch, um verschiedene Parametrierungen zu testen. Generell sind Mikrobenchmarks mit Vorsicht zu genießen [Kabutz06]: So könnte der Garbage Collector (GC) für Messabweichungen sorgen, da unser Integer-Array größer als 512 KB ist und somit beim nächsten vollständigen GC-Lauf in den Old Space ausgelagert wird. Außerdem kann sich der Java-HotSpot-Compiler einschalten, wenn er einen Hot-Spot entdeckt und durch Inlining optimiert. Solche Optimierungen müssen wir aus den Messungen gezielt herausrechnen. Da wir den HotSpot-Compiler nicht abschalten können, setzen wir das JVM-Argument **-XX:CompileThreshold=1**, damit der HotSpot-Compiler gleich bei der ersten Messung gezielt anspringt, und wiederholen dann den ersten Messlauf.

Nachstehende Ergebnisse einer Leistungsmessung basieren auf dieser Umgebung: i7-2700k, 3.5 GHz pro Kern, 4 Kerne mit Hyper-Threading. Wir berechnen ein Mandelbrot-Bild der Größe 2048x2048. Der kleinste Workload (eine Zeile des Bildes) berechnet 2048 Bildpunkte (vgl. auch Abb. 7):

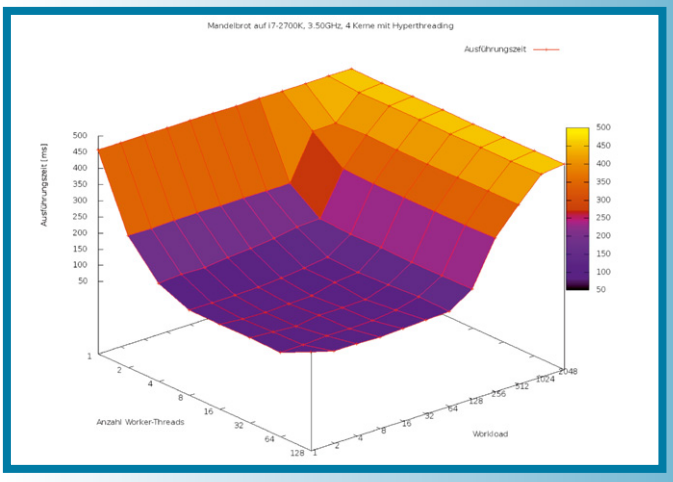

Abb. 7: Ausführungszeit in ms der Mandelbrot-Berechnung als Funktion der Parameter „Anzahl Worker-Threads“ (logarithmisch) und „Workload“ (logarithmisch)

- Ein optimales Ergebnis liefert eine Parametrierung von 8 oder 16 Worker-Threads bei einem Workload von 32 Bildzeilen pro Teilproblem.
- In den Randbereichen liegt eine quasi-sequenzielle Bearbeitung vor.
- Wir sehen eine deutliche Reduktion der Ausführungszeit bis zu Worker-Thread-Anzahl 32. Darüber hinaus konkurrieren zu viele Threads um Tasks (höhere Thread-Contention beim Zugriff auf die Deques), sodass sich die Ausführungszeit wieder erhöht.
- Bei einem zu großem Workload würde das Problem durch eine bessere Zerlegung in kleinere Arbeitspakete profitieren (Workload >= 256). Dieses Ergebnis ist daher unabhängig von der Worker-Thread-Anzahl (es bleibt „gleich schlecht“), sobald Worker-Thread-Anzahl <= (2048 / Workload).

- Der zugrunde liegenden tabellarischen Messreihe lässt sich entnehmen, dass sich bei steigender Worker-Thread-Anzahl auch die Varianz in den Messwerten erhöht (wegen mehr Thread-Contention). Die Varianz ab einer Anzahl von 64 Worker-Threads liegt bei ca. 18-20 ms, während sie bei Werten darunter bei ca. 1-2 ms liegt.

Wer tiefer einsteigen möchte und eine Innensicht des Fork-Join-Frameworks benötigt, dem kann ein neues Trace-Tool [fjp-trace] helfen, das als Open Source verfügbar ist. Es zeigt Instrumentierungsdetails der Threads bis hin zu Abhängigkeiten, Steals, Thread-Parks usw.

Speedup

Den Speedup berechnen wir, indem wir das Messergebnis mit einem Worker-Thread bei optimalem Workload (32 Zeilen) als Referenzwert mit Faktor 1,0 festlegen (s. Abb. 8). Passend zur Leistung der Testumgebung erreichen wir einen optimalen Speedup-Faktor bei knapp über 4.

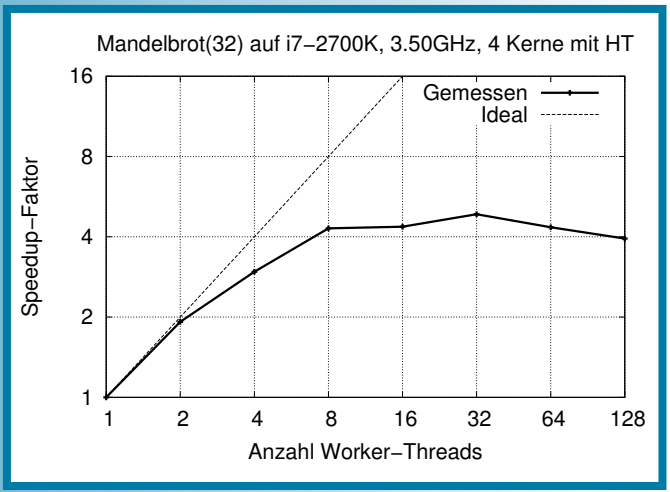

Abb. 8 (logarithmisch): Gemessen: Speedup der Mandelbrot-Berechnung als Funktion der Worker-Thread-Anzahl, ideal: Linearer Speedup von n (nur theoretisches Maximum wegen Amdahl's Law)

Fazit und Ausblick auf Java 8

Parallelisierung und effiziente Ausnutzung von Multicore-Plattformen wird ein immer wichtigeres Thema in der Anwendungsentwicklung: Wie implementiert man Parallelisierungsalgorithmen fehlerfrei, wie nutzt man sie effizient? Das Fork-Join-Framework in Java 7 eignet sich gut dafür, parallelisierbare, CPU-gebundene Tasks unter Zuhilfenahme eines Thread-Pools effizient zu lösen.

Mit dem für September 2013 angekündigten Java 8 wird das Fork-Join-Framework nochmals wichtiger und spannender, da es die Basis für parallelisierbare Collections-Operationen sein wird, insbesondere hinsichtlich des Einsatzes der neuen Lambda-Expressions und map/reduce/filter (s. JSR166e sowie JEP 103 in [jeps103] und JEP 107 in [jeps107]).

Literatur

[fjp-trace] `https://github.com/shipilev/fjp-trace`
[Goetz06] B. Goetz u. a., Java Concurrency in Practice, Addison-Wesley, 2006
[Gross12] D. Grossman, Beginners's Introduction to Java's ForkJoin Framework, `http://www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html`
[iX12] „iX Developer: Programmieren heute", Heft 1/2012. Verschiedene Beiträge zu Parallelprogrammierung, darunter auch: J. Link: „Kernschwemme – Grundsätzliches zur nebenläufigen und parallelen Programmierung"
[jeps103] JDK Enhancement-Proposal 103: Parallel Array Sorting, `http://openjdk.java.net/jeps/103`
[jeps107] JDK Enhancement-Proposal 107: Bulk Data Operations for Collections, `http://openjdk.java.net/jeps/107`
[JSR166] Java Specification Request 166: Concurrency Utilities, `http://jcp.org/en/jsr/detail?id=166`
[Kabutz06] H. G. Kabutz, Copying Arrays Fast, The Java Specialists' Newsletter, 28.3.2006, `http://www.javaspecialists.eu/archive/Issue124.html`
[Lea00] D. Lea, A Java Fork/Join Framework, in: JAVA '00 Proceedings of the ACM 2000 conference on Java Grande, S. 36-43, ACM New York, NY, USA, 2000
[Sutt05] H. Sutter, The Free Lunch Is Over, in: Dr. Dobb's Journal, März 2005, s. a. `http://www.gotw.ca/publications/concurrency-ddj.htm`

Quellen zum Beispiel:

http://www.sigs-datacom.de/nc/fachzeitschriften/javaspektrum/archiv.html

Markus Günther ist als Software-Ingenieur bei der Accelerated Solutions GmbH tätig und beschäftigt sich dort vornehmlich mit der Konzeption und Entwicklung Java-basierter Systeme.
E-Mail: guenther@accso.de

Martin Lehmann ist seit September 2010 als Cheftechnologe bei der Accelerated Solutions GmbH und in einem großen Java-Projekt für ein Konsortium internationaler Wetterdienste tätig.
E-Mail: lehmann@accso.de