Java im Multicore Zeitalter

Seminar "Programmiersprachen im Multicore-Zeitalter"

Ramon Ising

Inhaltsverzeichnis

1. Einleitung

Java ist eine objektorientierte Programmiersprache und befindet sich aktuell in Version 7. Die Java-Technologie besteht aus einem Java Development Kit (kurz JDK) und einer Java Runtime Enviroment (kurz JRE). Ein Teil der JRE ist die Java Virtual Machine (kurz JVM), auf dieser virtuellen Maschine werden die Programme ausgeführt. Durch diese Virtualisierung ist eine Plattformunabhängigkeit gegeben. In diesem Dokument wird auf die Parallelisierung mit Java eingegangen.

Die Thread Programmierung ist in Java sehr einfach möglich und solang das System auf einem Single-Core CPU läuft sind die Fehlerquellen überschaubar, da maximal 1 Thread wirklich gleichzeitig laufen kann. Sobald ein System aber über mehrere CPU-Cores verfügt, können mehrere Threads gleichzeitig laufen. In Java wird die Thread-Verwaltung in der Regel von dem Betriebssystem geregelt. Wenn dies der Fall ist, dann ist eine einfache Verteilung auf Mehrprozessorsysteme möglich. Das Problem hierbei besteht aber darin, dass alle Bibliotheken, die in den Threads aufgerufen werden, auch thread-sicher sein müssen. In den Anfängen gab es hier Probleme mit Unix-Versionen und den grafischen Standardbibliotheken X11 und Motif, diese waren nicht thread-sicher. [UL12]

Da bei der Parallelisierung von Programmen es vorkommen kann, dass zwei Threads auf die gleichen Ressourcen zugegriffen müssen, sind Fehler nicht ausgeschlossen. Daher bietet Java einige Möglichkeiten sicherzustellen, dass ein bestimmter Programmcode nur von einem Thread ausgeführt wird. Das einfachste Beispiel ist der synchronized Block, der aber auch zu Deadlocks[1] führen kann, daher gibt es noch andere Methoden, die in bestimmte Situationen besser sind.

2. Geschwindigkeitsvorteil

Wenn ein Programm parallel arbeitet, wird natürlich ein Geschwindigkeitsvorteil erwartet. Aber nicht immer tritt der gewünschte Effekt auf. Ein Beispiel wäre, wenn ein Programm aus 2 Threads besteht, aber der eine Thread immer auf den anderen Thread warten muss, tritt kein Geschwindigkeitsvorteil auf.

Hier ein Beispiel wo ein Geschwindigkeitsvorteil auftritt:

1. Datenbankverbindung aufbauen und Daten holen
2. Daten analysieren
3. Datei öffnen und Daten schreiben

Parallel könnte dies folgenermaßen aussehen:

1. Datenbankverbindung aufbauen und Datei öffnen kann parallel geschehen
2. Lesen neuer Datensätze und analysieren der alten Daten kann auch parallel geschehen
3. Alte analysierte Werte können schon in die Datei geschrieben werden, während noch neue Werte berechnet werden

[1] Deadlocks bedeutet Verklemmungen, durch diese Verklemmung kommt ein Programm aus einer bestimmten Funktion nicht mehr raus.

An dem Beispiel kann man schon sehr gut erkennen, dass hier eine bestimmte Synchronisation notwendig ist, damit das Programm ohne Probleme abläuft. Wenn z.B. schon neue Datensätze analysiert werden sollen, obwohl noch keine Datensätze da sind, muss erst gewartet werden. [UL12]

3. Thread erzeugen

Zum Einstieg werden jetzt die Möglichkeiten aufgezeigt, die Java besitzt einen Thread zu erzeugen und zu steuern. Hier wird noch keine Synchronisation beschrieben, sondern lediglich die „Grundfunktionen“. Da Java eine objektorientierte Programmiersprache ist, wird beim Erzeugen von Threads auf jeden Fall eine neue Klasse benötigt.

3.1. Schnittstelle „Runnable“

Die erste Möglichkeit besteht darin, eine neue Klasse zu erstellen und mit dieser Klasse das Interface „Runnable“ zu implementieren. In diesem Interface gibt es eine Methode „run“ und diese wird mit dem entsprechenden Programmcode implementiert. Wenn jetzt der Thread gestartet werden soll, wird die Klasse „Thread“ benötigt. Beim Initialisieren der Klasse Thread wird als Parameter die oben erstelle Klasse gegeben. Danach können wir den Thread mit der Methode „start“ erzeugen.

3.2. Klasse „Threads“

Bei der zweiten Möglichkeit wird von der Klasse „Thread“ geerbt und auch hier wird die Methode „run“ überschrieben. Der Vorteil bei dieser Variante ist aber, dass die neu erzeugte Klasse nur erstellt werden muss und danach sofort mit der Methode „start“ gestartet werden kann. Wir benötigen keine extra „Thread“ Klasse wie es bei „Runnable“ der Fall war.

4. Thread-Eigenschaften

Ein Thread hat verschiedene Eigenschaften und Funktionen, es werden hier ein paar davon aufgelistet:

Methode „getName“ und „setName“ liefert oder setzt den Namen eines entsprechenden Threads

Methode „getPriority“ und „setPriority“ liefert oder setzt die entsprechende Priorität eines Threads.

4.1. Thread.sleep

Mit der Methode „Thread.sleep“ ist es möglich einen Thread in einen Schlafzustand zu versetzen. Diesen Schlafzustand wird der Thread erst dann verlassen, wenn die entsprechende Zeit abgelaufen ist oder er von einem anderen Thread interrupted wird. Eine bestimmte Zeit warten, kann dann sinnvoll sein, wenn man weiß, dass ein bestimmtes Ereignis nicht sehr häufig auftritt. Auf Interrupts wird noch im folgenden Abschnitt eingegangen.

4.2. Join

Die Join-Methode kann eine Hilfreiche Funktion sein. Diese wartet solange, bis die Thread endgültig beendet ist.

4.3. Daemon[2]

Eine sehr interessante Methode lautet „setDaemon“. Hiermit kann man einen Thread erzeugen, der nur solange läuft, solang auch das Hautprogramm läuft. Dies kann für verschiedene Anwendungszwecke interessant sein, z.B. wenn ein Programm auf einen Dienst zurück greift, diesen aber nicht extra starten und stoppen möchte, sondern der immer verfügbar sein soll. Dann wird der Daemon am Anfang des Programmes gestartet und wenn das Programm beendet wird, wird der Thread automatisch mit beendet. Ein Problem gibt es hierbei, und zwar dann, wenn der Daemon auf die Festplatte Daten schreibt. Das kommt daher, weil wenn der Daemon gerade Daten auf die Festplatte schreibt und in diesem Moment das Programm beendet wird, dann darf der Daemon diese Dateien nicht zu Ende schreiben, sondern wird unterbrochen und das Programm beendet sich. Wenn man dann auf die Festplatte schaut, wird eine nicht komplett geschrieben Datei gefunden. Es müssen natürlich keine Daten auf der Festplatte sein, es könnten auch Daten im Netzwerk sein, die nicht erfolgreich geschrieben werden.

4.4. Interrupts:

Ein Thread kann mit einem Interrupt beendet werden, dafür muss der Thread mit der Methode „isInterrupted()“ prüfen, ob er beendet werden soll. Wenn ein Interrupt gesendet wird, unterbricht die Methode „sleep“ automatisch. Damit kann sichergestellt werden, dass der Thread sofort merkt, wenn er sich beenden soll.

Sehr hilfreich können Interrupts dann sein, wenn ein Thread in einer Endlosschleife auf bestimmte Ereignisse wartet. Diese können dann durch einen Interrupt beendet werden.

5. Java-Memory-Modell

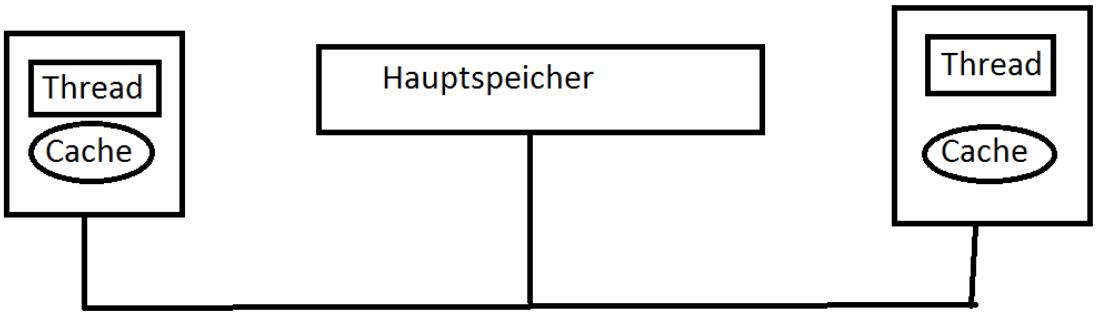


Abbildung 5.1

[2] Daemon ist ein Programm, was im Hintergrund läuft.

Das Java-Memory-Modell (kurz JMM) ist ähnlich wie auf Abbildung 5.1 zu sehen. Die Threads laufen parallel und haben alle Zugriff auf den Hauptspeicher, aber es ist zusehen, dass jeder seinen eigenen Cache besitzt, auf diesen Cache schreib ein Thread wenn er Variablen verändert. Wann und ob die Daten wieder in den Hauptspeicher geladen werden, dass entscheidet das JMM und daher gibt es bestimmte Regeln.

Eine Regel des JMM besagt, dass der Cache geladen wird, wenn ein Thread gestartet ist, dies kann man als selbstverständig ansehen. Eine weitere Selbstverständlichkeit ist es, dass wenn wir auf einen Thread warten (bis er beendet ist), die Änderungen die der Thread im Cache gemacht hat, in den Hauptspeicher übertragen werden.

Man sollte natürlich beachten, dass das JMM nur ein Modell ist, mit diesem Modell ist es allerdings möglich das Verwalten von einer JVM zu verstehen. In Wirklichkeit muss die JVM viel komplexere Operationen durchführen, da in einem CPU mehrere Caches existieren.

Ein kleines Beispiel:

Thread1 führt eine Aktion aus und setzt eine boolean Variable auf True

Thread2 wartet auf die boolean Variable und fängt danach erst an zu arbeiten.

Hierbei sollte es normalerweise keine Probleme geben und es wird auch häufig so gemacht, aber wenn man die Abbildung 1.1 sieht, dann wird klar, jeder Thread hat seinen eigenen Cache und ob die boolean Variable in den Hauptspeicher geschrieben wird, kann nicht gesagt werden. Es könnte also theoretisch passieren, dass Thread2 die Variable immer aus dem Cache sieht und niemals den neuen Variablen Inhalt zur Gesicht bekommen.

Das Problem kann mit einer Volatile-Variable gelöst werden, hierfür gibt es eine Garantie von der JMM, dass Thread2 den zuletzt geschrieben Wert liest.

5.1 Sichtbarkeit Regeln im JMM

Das Java-Memory-Modell regelt folgende Dinge:

- Atomicity
- Ordering
- Visibility

Bei der Atomicity geht es um die Ununterbrechbarkeit, d.h. dass ein Zugriff auf eine Variablen von einem primitivem Typ atomar sind, diese können nicht unterbrochen werden. Zwei Ausnahmen gibt es, das sind die Variablentypen Double und Long, diese sind erst als volatile Variablen atomar. Wenn man jetzt noch die Operationen auf diese Variablen atomar machen möchte, dann kann dazu auf das Packet „java.util.concurrent.atomic“ zurückgegriffen werden

Ordering ist ein sehr komplexes Thema, es geht hierbei darum, dass wenn ein Thread verschiedene Variablen schreibt, dass ein andere Thread genau die gleiche Reihenfolge wieder erkennt, wie diese Variablen geschrieben worden sind.

Ein Beispiel:

Thread1 schreibt Variable „a“, dann Variable „b“ und als letztes Variable „c“

Wenn jetzt das Ordering funktioniert kann Thread 2 erkennen, dass erst „a“ geschrieben worden ist und dann „b“ und dann „c“

Wenn man von Visibility redet, geht es darum, wenn ein Thread eine Modifikation am Speicher vornimmt, wann dies für einen anderen Thread sichtbar wird. Hier gibt das JMM eine Reihe von Garantien:

5.1.1. Threadstart- und ende:

Wenn ein Thread gestartet wird, dann wird automatisch ein Refresh des lokalen Arbeitsspeichers des Threads durchgeführt. D.h. ein Thread hat zum Start auf jeden Fall die richtigen Werte, wie diese auch im Hauptspeicher stehen. Das Gleiche gilt sobald der Thread beendet wird, dann werden die Werte die der Thread in seinem lokalen Arbeitsspeicher geändert hat in den Hauptspeicher geschrieben.

5.1.2. Synchronisation:

Sobald ein Programm einen Synchronisationsblock erreicht wird ein Refresh ausgeführt, d.h. alle Variablen die der Thread im Cache hat werden nochmal neu vom Hauptspeicher geladen. Wenn der Synchronistationblock vorbei ist wird automatisch ein Flush ausgeführt. Das bedeutet, dass alle Variablen die in dieser Zeit geändert worden sind wieder in den Hauptspeicher geschrieben werden.

5.1.3. Lese- und Schreibzugriff auf volatile-Variablen:

Volatile-Variablen haben eine besondere Eigenschaft. Sobald eine Volatile-Variable gelesen wird, wird ein Refresh ausgeführt. Des Weiteren wird auch Flush ausgeführt, wenn auf eine Variable geschrieben wird.

Ein Beispiel:

Thread1 schreibt Variable „a“, „b“ und danach die Volatile-Variable „c“

Thread2 liest Volatile-Variable „c“ und sieht den neuen Inhalt von „c“, dann gibt es eine Garantie, dass auch „a“ und „b“ die richten Inhalte haben, obwohl diese keine Volatile-Variablen sind.

5.1.4. Lese- und Schreibzugriff von atomaren Variablen:

Hierbei geht es um die atomaren Variablen aus dem Packet „java.util.concurrent.atomic“. Diese Variablen müssen via Get-Methode gelesen werden und dabei wird ein Refresh ausgeführt, genauso sieht es bei der Set-Methode aus, hierbei wird ein Flush ausgelöst. Es gibt bei diesen atomaren-Variablen noch weitere Methoden die eine read-and-modify-Operation atomar machen.

5.1.5. Erstmaliger Lesezugriff auf final-Variablen.

Eine final-Variable wird spätestens im Konstruktor einer Klasse initialisiert. Wenn diese final-Variable initialisiert wird, löst dies einen partiellen Flush aus, dabei werden alle Objekte die in Abhängigkeit mit der final-Variable stehen in den Hauptspeicher zurückgeschrieben.

Wie man jetzt erkennen kann gibt es eine Reihe von Garantien auf die man sich beziehen kann. Diese benötigt man natürlich nur, wenn man nicht nur Synchronisation benutzten will. Hierbei ist zu sagen, dass die Synchronisation immer einer der schlechtesten Fälle ist ein Problem zu lösen, da die entsprechenden Programmteile dann sequentiell und nicht parallel laufen. [LA08b]

6. Beispiel zur Mulitcore Programmierung

```
3  class Counter {
4      private int c = 0;
5
6      public void increment() {
7          c++;
8      }
9
10     public void decrement() {
11         c--;
12     }
13
14     public int value() {
15         return c;
16     }
17
18 }
```

Abbildung 6.1: Beispiel 1 [OR12]

```
2
3  class SynchronizedCounter {
4      private int c = 0;
5
6      public synchronized void increment() {
7          c++;
8      }
9
10     public synchronized void decrement() {
11         c--;
12     }
13
14     public synchronized int value() {
15         return c;
16     }
17
18 }
19
```

Abbildung 6.2: Beispiel 2 [OR12]

```
4  import java.util.concurrent.atomic.AtomicInteger;
5
6  class AtomicCounter {
7      private AtomicInteger c = new AtomicInteger(0);
8
9      public void increment() {
10         c.incrementAndGet();
11     }
12
13     public void decrement() {
14         c.decrementAndGet();
15     }
16
17     public int value() {
18         return c.get();
19     }
20
21 }
22
```

Abbildung 6.3: Beispiel 3 [OR12]

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class SynchronizedCounter {
    volatile private int c = 0;
    final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        c++;
        lock.unlock();
    }

    public void decrement() {
        lock.lock();
        c--;
        lock.unlock();
    }

    public int value() {
        return c;
    }

}
```

Abbildung 6.4: Beispiel 4

6.1. Beispiel 1

In der Abbildung 6.1 wird eine normale Counter-Klasse gezeigt, wie diese in einem sequenziellen Programm zu finden sein könnte. Hier wird auf nichts Besonderes geachtet. Die Funktion „increment“ erhört die Variable c um eins und die Funktion decremnt verringert die Variable c um eins. Wenn diese Klasse jetzt in einem Multithread Programm ausgeführt wird, könnte es passieren, dass zwei Programme gleichzeitig increment bzw. decremt ausführen. Wenn dies geschient, dann wird in c das falsche Ergebnis gespeichert, da die Anweisung „c++“ bzw „c--„ eigentlich drei Anweisungen sind, nämlich als erstes wird der Wert von c aus dem Speicher geladen und dann wird der Wert erhört/verringert und zum Schluss wieder zurück in den Speicher geschrieben.

Als Beispiel eine einfache 2 Thread Anwendung führen gleichzeitig increment aus (c auf 5).

Thread1 holt sich den Wert von c (5).

Thread2 holt sich den Wert von c (5)

Thread1 erhört den Wert um 1 auf 6

Thread2 erhört den Wert um 1 auf 6

Beide schreiben in die Variable c den Wert 6, obwohl die increment Funktion zweimal ausgeführt wurde.

6.2. Beispiel 2

Das nächste Beispiel ist auf Abbildung 6.2 zu sehen. Hier ist die gleiche Klasse zu sehen, wie im ersten Fall. Der Unterschied besteht darin, dass alle Funktionen dieser Klasse synchronized sind. Das bedeutet, dass immer maximal eine Funktion dieser Klasse aufgerufen werden kann.

In diesem Beispiel gelten die gleichen Bedingung wie bei dem letzten Beispiel:

Thread 1 ruft increment auf und sperrt dadurch einen weiteren Aufruf.

Thread2 versucht increment aufzurufen, wird aber in einen Wartezustand versetzt, da Thread1 in diesem Zustand ist.

Thread1 führt die Funktion erfolgreich aus und verlässt den Synchronized-Block.

Dadurch wird Thread2 aus dem Warte Zustand geholt und führt die Funktion ebenfalls erfolgreich aus.

Java gibt bei einem Synchronzied-Block eine Garantie dafür, dass sich maximal 1 Prozess in diesem befindet. Aber dadurch kann es auch zu schweren Problemen kommen, z.B. dann wenn ein Programm den Synchronzied-Block nicht verlässt, dann wird ein anderer Thread komplett geblockte und kann nicht mehr weiter laufen. Ein Deadlock entsteht und das Programm hängt sich auf.

6.3. Beispiel 3

Da eine Blockade immer nur der schlechteste Fall in einem Programm ist, gibt es für solche Aufgaben einen Atomaren-Datentyp in Java (siehe Abbildung 6.3). Dieser Datentyp hat eine eigene Funktion für increment / decremnt. Diese Funktionen sind automatisch richtig synchronisiert und nicht blockierten. Alle Datentypen die es bei Java in dem Packet „java.util.concurrent.atomic“ gibt sind automatisch thread-sicher und kommen ohne zusätzliche Synchronisation aus.

6.4. Beispiel 4

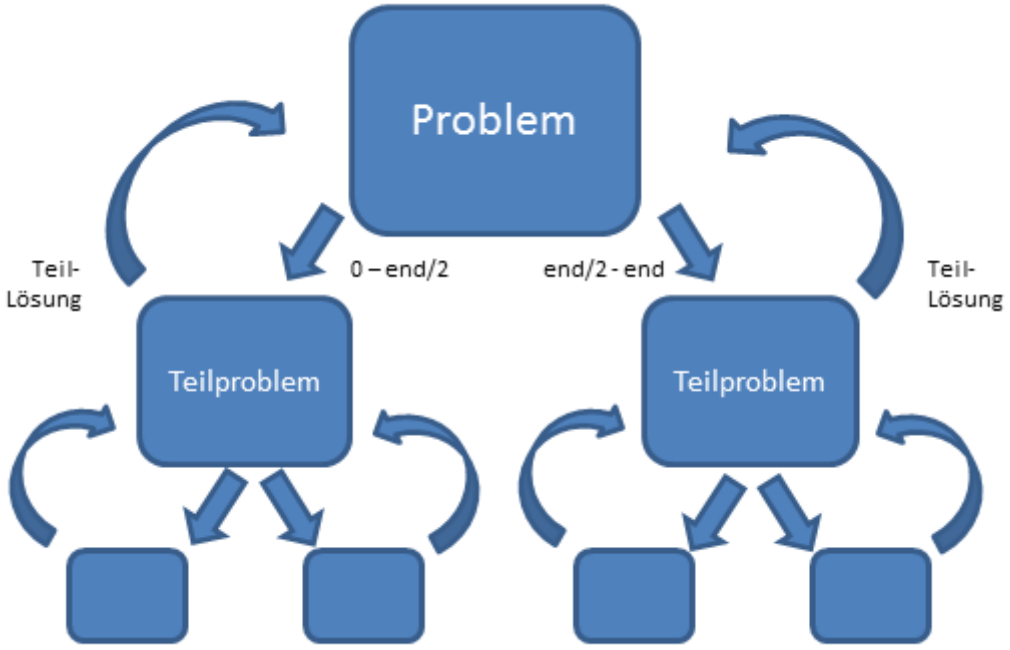
Das Beispiel 4 ist alternative zu Beispiel 2, hier wird nicht von Java die synchronized Funktionen genutzt, sondern der Counter wird manuell mit einem „ReentrantLock“ gesperrt. Der Vorteil dieser Klasse ist, dass diese mehr Funktionen bietet, z.B. gibt es eine Methode mit dem Namen „tryLock“. Diese legt nicht das komplette Programm lahm, weil wenn die Variable schon gesperrt wurde, kann dies abgefangen werden und dann kann in dieser Zeit eine andere Anweisung ausgeführt werden.

7. Performances

Wie man in den 4 Beispielen gesehen hat, gibt es verschiedene Wege ein Problem zu lösen. Beispiel 1 fällt natürlich raus, weil hier nicht die gewünschte Funktion bei einem Multithreading-Programm erzeugt wird. Bei Beispiel 2 existiert um jede Funktion ein synchronized-Block, dieser ist sehr Ineffizienz, da andere Threads warten müssen und keine weiteren Aktionen durchführen können. Wenn man dagegen Beispiel 4 anschaut, hier wird die Variable „c“ volatile gesetzt und dadurch wird ein synchronized-Block gespart. Aber diese Variante ist nicht so elegant wie Beispiel 3, dies ist die „perfekte“ Lösung für das Problem. Es werden keine Sperrelemente benutzt, weder Lock noch „synchronized“. Daher kann das Programm sofort weiter arbeiten und die atomare Variable muss sich intern selber um die Verarbeitung kümmern.

In der Entwicklung von Programmen sollte man auf jeden Fall versuchen auf synchronzied bzw. Lock-Elemente zu verzichten.

8. Fork/Join Framework



Abildung 8.1: Fork-Join Framework [EI11]

Seit Java 7 gibt es ein neues Fork/Join Framework. Mit Hilfe diesem Framework ist es noch einfacher bestimmte Aufgaben aufzuteilen. Das System ist sehr einfach zu bedienen.

Es gibt eine Klasse, die ein bestimmtes Problem lösen will, dieses Problem ist aber zu groß um Sequenziell ausgeführt zu werden. Daher teilt diese Klasse das Problem in 2 Teilprobleme auf und diese werden dann parallel gestartet und mit der invokeAll-Methode wieder zusammen gefügt. Den groben Ablauf kann man in Abbildung 8.1 erkennen. Hier wird das Problem in mehrere Teilprobleme aufgeteilt.

Um das Ganze ein wenig besser zu verdeutlichen, sieht man in Abbildung 8.2 ein Beispiel. Die Klasse „Solvler“ erbt von „RecursiveAction“, dadurch kann eine rekursive Funktion Parallel ausgeführt werden. In der compute-Methode wird dann die Rekursion sichtbar, solange das Array noch eine bestimmte Mindestgröße hat, werden die Funktionen mit „invokeAll“ aufgerufen, dadurch werden nachher die Threads automatisch erzeugt.

In der Abbildung 8.3 wird dann gezeigt, wie diese Methode dann parallel ausgeführt wird. Die Klasse „Solver“ und der „ForkJoinPool“ werden initialisiert. Der ForkJoinPool wird von dem neuen Java-Framework zur Verfügung gestellt. Diese Klasse erhält beim Erstellen die Anzahl an Thread die erstellt werden sollen. Danach bekommt der „ForkJoinPool“ den „Solver“ als Problem gegeben und das Framework kümmert sich automatisch um die Threadverwaltung.

```java
public class Solver extends RecursiveAction {
    private int[] list;
    public long result;

    private int lo;
    private int hi;
    final private int MINSIZE = 50000;

    public Solver(int[] array, int lo, int hi) {
        this.list = array;
        this.lo = lo;
        this.hi = hi;

    }

    @Override
    protected void compute() {
        int i, j;
        int schnitt;
        if (lo >= hi) {
            return;
        }
        schnitt = list[(lo+hi)/2];
        i = lo;
        j = hi;
        while (i <= j) {
            while (list[i] < schnitt)
                i++;
            while (list[j] > schnitt)
                j--;
            if (i <= j) {
                int tmp = list[i];
                list[i] = list[j];
                list[j] = tmp;
                i++;
                j--;
            }
        }

        int anzahl1 = j - lo;
        int anzahl2 = hi - i;
        if(anzahl1>MINSIZE || anzahl2>MINSIZE){
            Solver s1 = new Solver(list,lo,j);
            Solver s2 = new Solver(list,i,hi);
            invokeAll(s1,s2);
        }else{
            Solver s1 = new Solver(list,lo,j);
            Solver s2 = new Solver(list,i,hi);
            s1.compute();
            s2.compute();
        }


    }

    public int[] getList() {
        return list;
    }
}
```

Abbildung 8.2 Fork-Join Modell Beispiel

```
int[] liste = {4,8,1,2};
int nThreads = Runtime.getRuntime().availableProcessors();
Solver solv = new Solver(liste,0,liste.length-1);
ForkJoinPool pool = new ForkJoinPool(nThreads);
pool.invoke(solv);
```

(Abbildung 8.3 Fork-Join Modell Beispiel)

9. Concurrent ExecutorService / Callables

```
public class MyCallable implements Callable<Long> {
        @Override
        public Long call() throws Exception {
                long sum = 0;
                for (long i = 0; i <= 100; i++) {
                        sum += i;
                }
                return sum;
        }

}
```

Abbildung 9.1 Callable

```
public class CallableFutures {
        private static final int NTHREDS = 10;

        public static void main(String[] args) {

                ExecutorService executor = Executors.newFixedThreadPool(NTHREDS);
                List<Future<Long>> list = new ArrayList<Future<Long>>();
                for (int i = 0; i < 20000; i++) {
                        Callable<Long> worker = new MyCallable();
                        Future<Long> submit = executor.submit(worker);
                        list.add(submit);
                }
                long sum = 0;
                System.out.println(list.size());
                // Now retrieve the result
                for (Future<Long> future : list) {
                        try {
                                sum += future.get();
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        } catch (ExecutionException e) {
                                e.printStackTrace();
                        }
                }
                System.out.println(sum);
                executor.shutdown();
        }
}
```

Abbildung 9.2 ExecutorService

Eine weitere Möglichkeit mit mehrere Threads zu arbeiten, ist der „ExecutorService“. Um diesen zu nutzen, wird eine Klasse benötigt die von dem Interface „Callable“ ableitet. In dieser Klasse schreibt man das Problem, welches gelöst werden soll.

Wenn man den „ExeuctorService“ initialisiert, dann wird als Parameter die Anzahl von maximalen Threads übergeben. Der nächste Schritt ist es, dem ExecutorService seine „Arbeit“ zu übergeben, die wird in dem Beispiel (Abbildung 9.2) sehr gut deutlich. Es wird eine Klasse „MyCallable“ initialisiert und danach dem „ExeuctorService“ übergeben. Um die „Arbeit“ zu starten wird die Methode „submit“ genutzt, diese gibt als Rückgabewert eine Referenz, die man später wieder zum Sammeln der Daten nutzen kann. Der Vorteil hierbei ist es, dass man sich keine weiteren Gedanken um die Threadverwaltung machen muss, da diese von dem Framework übernommen wird.

Zum Schluss geht man alle Referenzen durch, die wir von dem ExecutorService haben und ruft die Werte mit der Get-Methode ab. In dem Beispiel auf Abbildung 9.2 werden alle Werte zu einer Summe addiert.

10. Fazit

Wenn man über das ganze Thema einmal blickt, erkennt man schnell, dass es schon ein paar Möglichkeiten gibt, ein Programm in Java einfach zu parallelisieren.

Gerade wenn man mit einem „Fork/Join-Modell“ oder einem „ExecutorService“ arbeitet, muss man sich als Programmierer um nicht viel kümmern. Aber auch die „normale“ Parallelisierung mit Java und der Thread-Klasse schon sehr einfach möglich. Hier kann man sich auf das Framework verlassen.

Wenn man noch einmal auf das Java-Memory-Modell schaut, sieht man, dass hier nur ein paar Regeln eingehalten werden müssen und dafür werden Garantien geschaffen.

Es sollte auf jeden Fall immer darauf geachtet werden, dass wenn möglich die Volatile-Variablen oder atomaren Variablen benutzt werden. Hiermit können sehr gute Ergebnisse mit erzielt werden. Das Problem bei Volitale-Variablen ist leider, dass der Code etwas komplizierter werden kann, gerade wenn es um größere Probleme geht, die sehr viel Parallelisierung und Kommunikation benötigen.

11. Literaturverzeichnis

[Com12a] Wikipedia Community. Java http://de.wikipedia.org/wiki/Java_%28Programmiersprache%29

[VO12] Lars Vogel http://www.vogella.de/articles/JavaConcurrency/article.html

[LA08a] Angelika Langer http://www.angelikalanger.com/Articles/EffectiveJava/37.JMM-Introduction/37.JMM-Introduction.html

[LA08b] Angelika Langer http://www.angelikalanger.com/Articles/EffectiveJava/38.JMM-Overview/38.JMM-Overview.html

[LA09] Angelika Langer http://it-republik.de/jaxenter/news/JAX-TV-Java-Programmierung-im-Multicore-Zeitalter-047030.html

[UL12] Christian Ulleboom http://openbook.galileocomputing.de/javainsel/

[WE02] Mike Werner http://it-republik.de/jaxenter/artikel/Threads-in-Java-0228.html

[OR12] Orcale http://docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html

[EI11] Markus Eisele http://www.heise.de/developer/artikel/Was-ist-neu-in-Java-7-Teil-2-Performance-1288272.html

[WI12] Roland Wismüller Uni Siegen http://www.bs.informatik.uni-siegen.de/lehre/ws1112/pv/index_html