1 Einführung

Die meisten Computer können heute verschiedene Anweisungen parallel abarbeiten. Um diese zur Verfügung stehende Ressource auszunutzen, müssen wir sie bei der Softwareentwicklung entsprechend berücksichtigen. Die nebenläufige Programmierung wird deshalb häufiger eingesetzt. Der Umgang und die Koordinierung von *Threads* gehören heute zum Grundhandwerk eines guten Entwicklers.

1.1 Dimensionen der Parallelität

Bei Softwaresystemen gibt es verschiedene Ebenen, auf denen Parallelisierung eingesetzt werden kann bzw. bereits eingesetzt wird. Grundlegend kann zwischen Parallelität auf der Prozessorebene und der Systemebene unterschieden werden [26, 15]. Auf der Prozessorebene lassen sich die drei Bereiche *Pipelining* (Fließbandverarbeitung), superskalare Ausführung und Vektorisierung für die Parallelisierung identifizieren.

Auf der Systemebene können je nach Prozessoranordnung und Zugriffsart auf gemeinsam benutzte Daten folgende Varianten unterschieden werden:

- Bei *Multinode-Systemen* wird die Aufgabe über verschiedene Rechner hinweg verteilt. Jeder einzelne Knoten (in der Regel ein eigenständiger Rechner) hat seinen eigenen Speicher und Prozessor. Man spricht in diesem Zusammenhang von verteilten Anwendungen.
- Bei *Multiprocessor-Systemen* ist die Anwendung auf verschiedene Prozessoren verteilt, die sich in der Regel alle auf demselben Rechner (Mainboard) befinden und die alle auf denselben Hauptspeicher zugreifen, wobei die Zugriffszeiten nicht einheitlich sind. Jeder Prozessor hat darüber hinaus auch noch verschiedene Cache-Levels. Solche Systeme besitzen häufig eine sogenannte NUMA-Architektur (*Non-Uniform Memory Access*).
- Bei *Multicore-Systemen* befinden sich verschiedene Rechenkerne in einem Prozessor, die sich den Hauptspeicher und zum Teil auch Caches teilen. Der Zugriff auf den Hauptspeicher ist von allen Kernen

gleich schnell. Man spricht in diesem Zusammenhang von einer UMA-Architektur (*Uniform Memory Access*).

Neben den hier aufgeführten allgemeinen Unterscheidungsmerkmalen gibt es noch weitere, herstellerspezifische Erweiterungsebenen. Genannt sei hier z. B. das von Intel eingeführte Hyper-Threading. Dabei werden Lücken in der Fließbandverarbeitung mit Befehlen von anderen Prozessen möglichst aufgefüllt.

Hinweis

In dem vorliegenden Buch werden wir uns ausschließlich mit den Konzepten und Programmiermodellen für Multicore- bzw. Multiprocessor-Systeme mit Zugriff auf einen gemeinsam benutzten Hauptspeicher befassen, wobei wir auf die Besonderheiten der NUMA-Architektur nicht eingehen. Bei Java hat man außer der Verwendung der beiden VM-Flags `-XX:+UseNUMA` und `-XX:+UseParallelGC` kaum Einfluss auf das Speichermanagement.

1.2 Parallelität und Nebenläufigkeit

Zwei oder mehrere Aktivitäten (*Tasks*) heißen *nebenläufig*, wenn sie zeitgleich bearbeitet werden können. Dabei ist es unwichtig, ob zuerst der eine und dann der andere ausgeführt wird, ob sie in umgekehrter Reihenfolge oder gleichzeitig erledigt werden. Sie haben keine kausale Abhängigkeit, d.h., das Ergebnis einer Aktivität hat keine Wirkung auf das Ergebnis einer anderen und umgekehrt. Das Abstraktionskonzept für Nebenläufigkeit ist bei Java der *Thread*, der einem eigenständigen Kontrollfluss entspricht.

Besitzt ein Rechner mehr als eine CPU bzw. mehrere Rechenkerne, kann die Nebenläufigkeit parallel auf Hardwareebene realisiert werden. Dadurch besteht die Möglichkeit, die Abarbeitung eines Programms zu beschleunigen, wenn der zugehörige Kontrollfluss nebenläufige Tasks (Aktivitäten) beinhaltet. Dabei können moderne Hardware und Übersetzer nur bis zu einem gewissen Grad automatisch ermitteln, ob Anweisungen sequenziell oder parallel (gleichzeitig) ausgeführt werden können. Damit Programme die Möglichkeiten der Multicore-Prozessoren voll ausnutzen können, müssen wir die Parallelität explizit im Code berücksichtigen.

Die nebenläufige bzw. parallele Programmierung beschäftigt sich zum einen mit Techniken, wie ein Programm in einzelne, nebenläufige Abschnitte/Teilaktivitäten zerlegt werden kann, zum anderen mit den verschiedenen Mechanismen, mit denen nebenläufige Abläufe synchronisiert und gesteu-

ert werden können. So schlagen z. B. Mattson et al. in [37] ein »patternbasiertes« Vorgehen für das Design paralleler Anwendungen vor. Ähnliche Wege werden auch in [7] oder [38] aufgezeigt. Spezielle Design-Patterns für die nebenläufige Programmierung findet man in [15, 38, 42, 45].

1.2.1 Die Vorteile von Nebenläufigkeit

Der Einsatz von Nebenläufigkeit ermöglicht die Anwendung verschiedener neuer Programmierkonzepte. Der offensichtlichste Vorteil ist die Steigerung der Performance. Auf Maschinen mit mehreren CPUs kann zum Beispiel das Sortieren eines großen Arrays auf mehrere Threads verteilt werden. Dadurch kann die zur Verfügung stehende Rechenleistung voll ausgenutzt und somit die Leistungsfähigkeit der Anwendung verbessert werden. Ein weiterer Aspekt ist, dass Threads ihre Aktivitäten unterbrechen und wiederaufnehmen können. Durch Auslagerung der blockierenden Tätigkeiten in separate Threads kann die CPU in der Zwischenzeit andere Aufgaben erledigen. Hierdurch ist es möglich, asynchrone Schnittstellen zu implementieren und somit die Anwendung reaktiv zu halten. Dieser Gesichtspunkt gewinnt immer mehr an Bedeutung.

1.2.2 Die Nachteile von Nebenläufigkeit

Der Einsatz von Nebenläufigkeit hat aber nicht nur Vorteile. Er kann unter Umständen sogar mehr Probleme verursachen, als damit gelöst werden. Programmcode mit Multithreading-Konzepten ist nämlich oft schwer zu verstehen und mit hohem Aufwand zu warten. Insbesondere wird das Debugging erschwert, da die CPU-Zuteilung an die Threads nicht deterministisch ist und ein Programm somit jedes Mal verschieden verzahnt abläuft.

Parallel ablaufende Threads müssen koordiniert werden, sodass man immer mehrere Programmflüsse im Auge haben muss, insbesondere wenn sie auf gemeinsame Daten zugreifen. Wenn eine Variable von einem Thread geschrieben wird, während der andere sie liest, kann das dazu führen, dass das System in einen falschen Zustand gerät. Für gemeinsam verwendete Objekte müssen gesondert Synchronisationsmechanismen eingesetzt werden, um konsistente Zustände sicherzustellen. Des Weiteren kommen auch Cache-Effekte hinzu. Laufen zwei Threads auf verschiedenen Kernen, so besitzt jeder seine eigene Sicht auf die Variablenwerte. Man muss nun dafür Sorge tragen, dass gemeinsam benutzte Daten, die aus Performance-Gründen in den Caches gehalten werden, immer synchron bleiben. Weiter ist es möglich, dass sich Threads gegenseitig in ihrem Fortkommen behindern oder sogar verklemmen.

1.2.3 Sicherer Umgang mit Nebenläufigkeit

Den verschiedenen Nachteilen versucht man durch die Einführung von Parallelisierungs- und Synchronisationskonzepten auf höherer Ebene entgegenzuwirken. Ziel ist es, dass Entwickler möglichst wenig mit *Low-Level*-Synchronisation und Thread-Koordination in Berührung kommen. Hierzu gibt es verschiedene Vorgehensweisen. So wird z. B. bei C/C++ mit OpenMP[1] die Steuerung der Parallelität deklarativ über `#pragma` im Code verankert. Der Compiler erzeugt aufgrund dieser Angaben parallel ablaufenden Code. Die Sprache Cilk erweitert C/C++ um neue Schlüsselworte, wie z. B. `cilk_for`[2].

Java geht hier den Weg über die Bereitstellung einer »*Concurrency*-Bibliothek«, die mit Java 5 eingeführt wurde und sukzessive erweitert wird. Nachdem zuerst Abstraktions- und Synchronisationskonzepte wie *Threadpools*, *Locks*, *Semaphore* und *Barrieren* angeboten wurden, sind mit Java 7 und Java 8 auch Parallelisierungsframeworks hinzugekommen. Nicht vergessen werden darf hier auch die Einführung Thread-sicherer Datenstrukturen, die unverzichtbar bei der Implementierung von Multithreaded-Anwendungen sind. Der Umgang mit diesen *High-Level*-Abstraktionen ist bequem und einfach. Nichtsdestotrotz gibt es auch hier Fallen, die man nur dann erkennt, wenn man die zugrunde liegenden *Low-Level*-Konzepte beherrscht. Deshalb werden im ersten Teil des Buches die Basiskonzepte ausführlich erklärt, auch wenn diese im direkten Praxiseinsatz immer mehr an Bedeutung verlieren.

1.3 Maße für die Parallelisierung

Neben der Schwierigkeit, korrekte nebenläufige Programme zu entwickeln, gibt es auch inhärente Grenzen für die Beschleunigung durch Parallelisierung. Eine wichtige Maßzahl für den Performance-Gewinn ist der *Speedup* (Beschleunigung bzw. Leistungssteigerung), der wie folgt definiert ist:

$$S = \frac{T_{seq}}{T_{par}}$$

Hierbei ist T_{seq} die Laufzeit mit einem Kern und T_{par} die Laufzeit mit mehreren.

1.3.1 Die Gesetze von Amdahl und Gustafson

Eine erste Näherung für den *Speedup* liefert das Gesetz von Amdahl [2]. Hier fasst man die Programmteile zusammen, die parallel ablaufen können.

[1] Siehe `http://www.openmp.org`.

[2] Siehe `http://www.cilkplus.org`.

Wenn P der prozentuale, parallelisierbare Anteil ist, dann entspricht $(1-P)$ dem sequenziellen, nicht parallelisierbaren. Hat man nun N Prozessoren bzw. Rechenkerne zur Verfügung, so ergibt sich der maximale *Speedup*

$$S(N) = \frac{\text{Sequenzielle Laufzeit}}{\text{Parallele Laufzeit}} = \frac{1}{\frac{P}{N} + (1-P)},$$

wobei hier implizit davon ausgegangen wird, dass die Parallelisierung einen konstanten, vernachlässigbaren, internen Verwaltungsaufwand verursacht. Durch Grenzwertbildung $N \to \infty$ ergibt sich dann der theoretisch maximal erreichbare *Speedup* beim Einsatz von unendlich vielen Kernen bzw. Prozessoren zu

$$\lim_{N\to\infty} S(N) = \lim_{N\to\infty} \frac{1}{\frac{P}{N} + (1-P)} = \frac{1}{(1-P)}.$$

An der Formel sieht man, dass der nicht parallelisierbare Anteil den *Speedup* begrenzt. Beträgt der parallelisierbare Anteil z.B. nur 50%, so kann nach dem Amdahl'schen Gesetz maximal nur eine Verdopplung der Ausführungsgeschwindigkeit erreicht werden (vgl. Abb. 1-1).

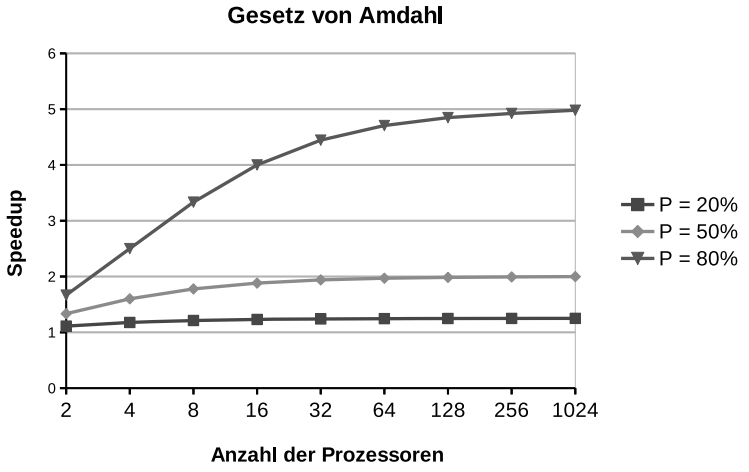

Abbildung 1-1: *Speedup* in Abhängigkeit von P und N

Man kann die Parallelisierung aber auch unter einem anderen Gesichtspunkt betrachten. Amdahl geht von einem fest vorgegebenen Programm bzw. einer fixen Problemgröße aus. Gustafson betrachtet dagegen eine variable Problemgröße in einem festen Zeitfenster [18]. Er macht die Annahme, dass sich die Vergrößerung des zu berechnenden Problems im Wesentlichen üblicherweise nur auf den parallelisierbaren Programmteil P auswirkt (man sagt, die Anwendung ist skalierbar). Unter diesem Aspekt er-

gibt sich ein *Speedup* von

$$S(N) = (1 - P) + N \cdot P$$

d.h., der Zuwachs ist hier proportional zu N.

Die unterschiedlichen Sichtweisen zwischen Amdahl und Gustafson sind in der Abbildung 1-2 verdeutlicht.

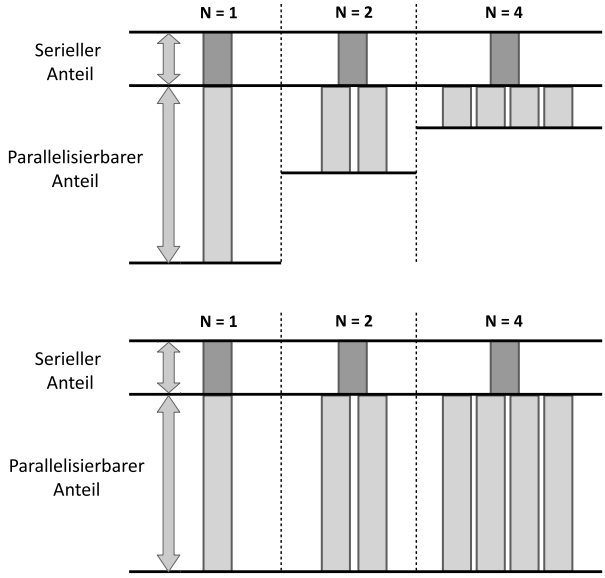

Abbildung 1-2: Amdahl (oben) versus Gustafson (unten)

1.3.2 Work-Span-Analyse

Eine weitere Methode, den Grad einer Parallelisierung zu beschreiben, ist die *Work-Span*-Analyse [10]. In dem zugrunde liegenden Modell werden die Abhängigkeiten der auszuführenden Aktivitäten in einem azyklischen Graphen dargestellt (vgl. Abb. 1-3). Eine Aktivität kann hier erst dann ausgeführt werden, wenn alle »Vorgänger« abgeschlossen sind.

Die von dem Algorithmus zu leistende Gesamtarbeit ist die Summe der auszuführenden Aktivitäten. Man bezeichnet die benötigte Zeit (*work*) hierfür mit T_1. Der sogenannte *span*, der mit T_∞ bezeichnet wird, entspricht dem kritischen Pfad, also dem längsten Weg von Aktivitäten, die nacheinander ausgeführt werden müssen[3].

[3] In der Literatur wird der *span* auch manchmal als *step complexity* oder *depth* bezeichnet.

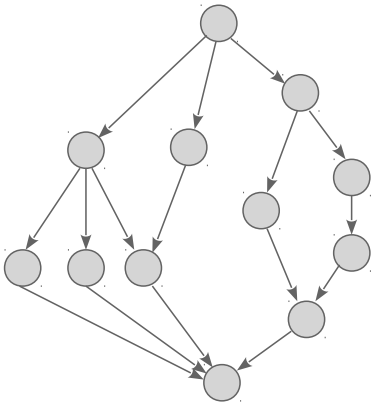

Abbildung 1-3: Azyklischer Aktivitätsgraph

Wenn wir uns den Aktivitätsgraphen in Abbildung 1-3 anschauen und annehmen, dass jede Aktivität eine Zeiteinheit dauert, so erhalten wir für den *work* $T_1 = 12$ und den *span* $T_\infty = 6$. Sei N wieder die Anzahl der Rechenkerne bzw. Prozessoren, dann erhält man als Speedup:

$$S(N) = \frac{T_1}{T_N} \leq N.$$

Der Speedup wächst linear mit der Anzahl der Prozessoren, vorausgesetzt dass die CPU immer voll ausgelastet ist (*greedy scheduling*). Der Speedup ist allerdings durch den *span* begrenzt, da der kritische Pfad sequenziell abgearbeitet werden muss:

$$S(N) = \frac{T_1}{T_N} \leq \frac{T_1}{T_\infty} = \frac{work}{span}.$$

In unserem Beispiel beträgt der maximal erreichbare Speedup $T1/T_\infty = 2$.

1.4 Parallelitätsmodelle

In der Literatur wird zwischen verschiedenen Modellen für die Parallelisierung unterschieden. Java unterstützt jedes dieser Modelle durch das Bereitstellen verschiedener Konzepte und APIs.

Zur Parallelisierung von Anwendungen gibt es grundsätzlich zwei Ansätze: Daten- und Task-Parallelität[4]. Bei der *Datenparallelität* wird ein Datenbestand geteilt und die Bearbeitung der Teilbereiche verschiedenen Threads zugeordnet. Hierbei führt jeder Thread dieselben Operationen aus.

[4]Die beiden Parallelisierungskonzepte werden ausführlich in [15] diskutiert.

Diese Art der Parallelisierung wird durch das Gesetz von Gustafson beschrieben und ist in der Regel gut skalierbar [53]. Mit dem ForkJoin-Framework und dem `Stream`-API stehen bei Java hierfür zwei leistungsfähige Möglichkeiten zur Verfügung (siehe Kapitel 13 und 14). Falls man diese Frameworks nicht einsetzen möchte, kann für eine explizite Umsetzung auf zahlreiche Synchronisationskonzepte zurückgegriffen werden (siehe Kapitel 11 und 12).

Bei der *Task-Parallelität*[5] wird die Anwendung in Funktionseinheiten zerlegt, die dann bezüglich ihrer Abhängigkeiten ausgeführt werden. Diese Art der Parallelisierung wird durch die *Work-Span*-Analyse beschrieben und kann bei Java mithilfe der `CompletableFuture`-Klasse oder je nachdem auch mit dem ForkJoin-Framework realisiert werden (siehe Kapitel 13 und 15).

Neben diesen beiden grundsätzlichen Ansätzen wird auch oft noch zwischen dem *Master-Slave*-, dem *Work-Pool*- und dem *Erzeuger-Verbraucher*- bzw. *Pipeline*-Programmiermuster unterschieden [32]. Das Unterscheidungsmerkmal ist hierbei die Art und Weise, wie die beteiligten Komponenten miteinander kommunizieren. Beim *Master-Slave*-Modell gibt es einen dedizierten Thread, der Aufgaben an andere verteilt und dann die Ergebnisse einsammelt. Bei Java kann dieses Modell mit dem `Future`-Konzept umgesetzt werden (siehe Abschnitt 6.2). Das *Work-Pool*-Modell entspricht dem `ExecutorService`, dem man Aufgaben zur Ausführung delegieren kann (siehe Abschnitt 6.1). Das bewährte *Erzeuger-Verbraucher*-Modell wird typischerweise durch `BlockingQueue`-Datenstrukturen realisiert und existiert in verschiedenen Varianten (siehe Abschnitt 10.3). In der Praxis findet man häufig Kombinationen der verschiedenen Modelle bzw. Muster.

[5] Genauer müsste man eigentlich »funktionale Dekomposition« (*functional decomposition*) sagen, da der Begriff Task-Parallelität oft auf alles Mögliche angewendet wird.

13 Das ForkJoin-Framework

Das ForkJoin-Framework wurde mit Java 7 eingeführt und kann insbesondere für die Parallelisierung von *Divide-and-Conquer*-Algorithmen eingesetzt werden. Es verwendet intern einen Threadpool, der ein *Work-Stealing*-Verfahren implementiert, das dafür sorgt, dass die verfügbaren Rechenressourcen optimal ausgenutzt werden. Das ForkJoin-Framework realisiert das aus der Literatur bekannte *ForkJoin-Pattern* [15, 34, 37, 38].

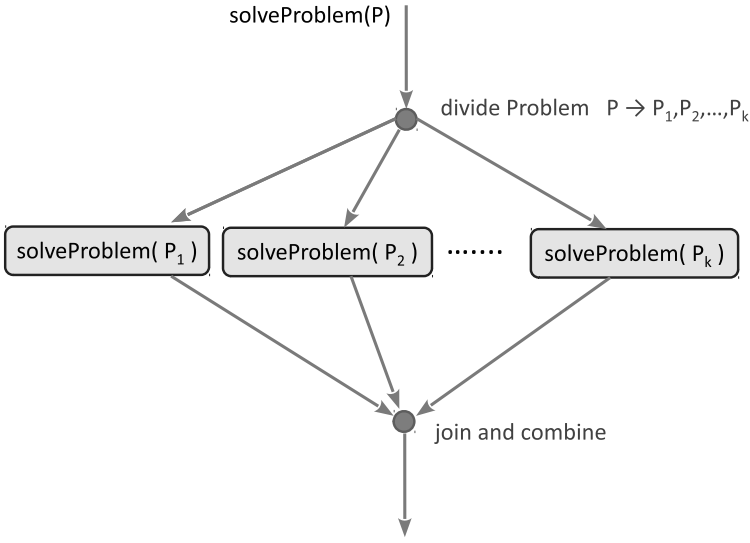

Abbildung 13-1: Der Kontrollfluss des ForkJoin-Patterns

13.1 Grundprinzip des ForkJoin-Patterns

Beim ForkJoin-Pattern wird der Kontrollfluss an einer dedizierten Stelle in mehrere nebenläufige Flüsse aufgeteilt (*fork*), die an einer späteren Stelle alle wieder vereint (*join*) werden (vgl. Abb. 13-1). Die Vereinigung entspricht

einem Synchronisationspunkt. Wenn alle Teilaufgaben erledigt sind, wird das Programm danach fortgesetzt.

Die Stärke bzw. die eigentliche Anwendung des ForkJoin-Patterns tritt bei der Umsetzung rekursiver *Divide-and-Conquer*-Algorithmen zutage. Ein typisches Programm-Muster ist im Algorithmus 1 zu sehen.

Algorithmus 1 Pseudocode für den Einsatz des ForkJoin-Patterns

```
function SOLVEPROBLEM(Problem P)
    if P.size < THRESHOLD then
        solve P sequentially
    else
        divide P in k subproblems P_1, P_2, ..., P_k
        ▷ fork to conquer each subproblem in parallel
        fork solveProblem(P_1)
        fork solveProblem(P_2)
        fork ...
        fork solveProblem(P_k)
        join
    end if
end function
```

Abbildung 13-2 zeigt schematisch die rekursive Verzweigungs- und Vereinigungsstruktur. In der ersten Phase wird das Problem immer wieder zerkleinert (*Divide*-Phase). Ist eine entsprechende Problemgröße erreicht, werden die Teilaufgaben gelöst (*Work*-Phase) und anschließend das Ergebnis zusammengesetzt (*Combine*-Phase).

13.2 Programmiermodell

Die zentralen Komponenten des ForkJoin-Frameworks bestehen aus dem `ForkJoinPool`-Threadpool und den von `ForkJoinTask` abgeleiteten abstrakten Klassen `RecursiveAction`, `RecursiveTask` und `CountedCompleter` (vgl. Abb. 13-3). Die Basisklasse für Tasks ohne Rückgabe ist `RecursiveAction`. Soll ein Wert zurückgeliefert werden, müssen die Tasks von der Klasse `RecursiveTask` ableiten. Der bei Java 8 neu hinzugekommene `CountedCompleter` kann benutzt werden, wenn man z. B. das Warten auf das Ende der Sub-Tasks selbst steuern möchte.

Der `ForkJoinPool` wurde bereits in Abschnitt 6.5 kurz vorgestellt. Er besitzt die Konstruktoren `ForkJoinPool()`, `ForkJoinPool(int parallelism)` und einen, bei dem explizit eine `ThreadFactory`, ein `UncaughtExceptionHandler` und der Ausführungsmodus angegeben

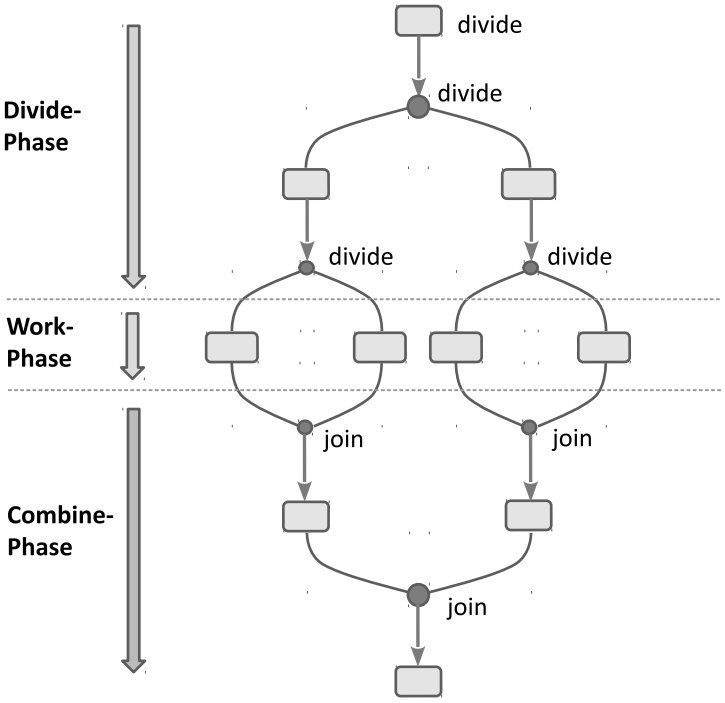

Abbildung 13-2: Rekursive Verwendung des ForkJoin-Patterns

werden. Die für den Umgang mit dem ForkJoin-Framework wichtigen Methoden sind:

- `void execute(ForkJoinTask<?> task)`: Führt den übergebenen Task asynchron aus.
- `T invoke(ForkJoinTask<T> task)`: Startet die Ausführung des Tasks, wobei gewartet wird, bis er fertig ist (synchrone Ausführung).
- `ForkJoinTask<T> submit(ForkJoinTask<T> task)`: Führt den übergebenen Task asynchron aus und liefert ein `ForkJoinTask`-Objekt zurück, das auch ein `Future` ist und mit dem man z. B. auf den Rückgabewert zugreifen kann.

Die von den Tasks zu implementierende Methode ist `compute`, in der die Aufteilung des Problems und die Verzweigung in die Teilprobleme durchgeführt wird (vgl. Algorithmus 1).

Für die Verzweigung stehen die Methoden `fork` und `invoke` zur Verfügung. Mit `fork` wird die asynchrone, nicht blockierende Ausführung des Tasks gestartet. Dagegen wartet `invoke` blockiert, bis alle Teilaufgaben erledigt sind. Mit `join` kann das Ergebnis der Verarbeitung abgeholt werden.

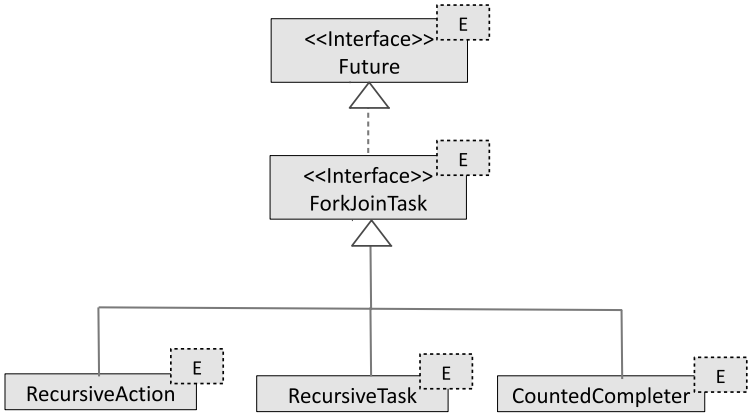

Abbildung 13-3: Hierarchie der `Task`-Klassen

Die von `Future` geerbte Methode `get` verhält sich wie `join`, wirft aber im Fehlerfall eine `InterruptedException` oder `ExecutionException`.

Tabelle 13-1 listet die gebräuchlichen Methoden auf, wobei unterschieden wird, wann und wo sie verwendet werden können. Die Methoden `execute, invoke, submit` dienen als Startpunkte. Dagegen werden `fork` und `invoke` innerhalb der `compute`-Methode aufgerufen und realisieren somit rekursive asynchrone bzw. synchrone Aufrufe.

	Aufruf außerhalb eines ForkJoin-Tasks	Aufruf innerhalb eines ForkJoin-Tasks
Asynchrone Ausführung	`execute(ForkJoinTask)`	`ForkJoinTask.fork()`
Synchrone Ausführung (blockierend)	`invoke(ForkJoinTask)`	`ForkJoinTask.invoke()`
Asynchrone Ausführung, Rückgabewert über Future-Objekt	`submit(ForkJoinTask)`	`ForkJoinTask.fork()`

Tabelle 13-1: Wichtige Methoden des ForkJoin-Frameworks

13.2.1 Einsatz von RecursiveAction

Beim Einsatz des ForkJoin-Frameworks findet man im Prinzip immer ein ähnliches Code-Template. Codebeispiel 13.1 zeigt ein `RecursiveAction`-Objekt, das je nach Fall in drei Sub-Tasks verzweigt. Die Methode `invokeAll` blockiert und kehrt erst zurück, wenn alle Teilaufgaben beendet sind (❶).

```
public class SimpleTask extends RecursiveAction
{
  // Member-Variablen
  // Konstruktoren

  @Override
  protected void compute()
  {
    if( ... )
    {
      // Serieller Algorithmus
    }
    else
    {
      // Definition von drei Sub-Tasks
      SimpleTask task1 = new SimpleTask(...);
      SimpleTask task2 = new SimpleTask(...);
      SimpleTask task3 = new SimpleTask(...);
      // task1, task2 und task3 werden asynchron ausgeführt
      invokeAll(task1,task2,task3);                                    ❶
    }
  }
}
```

Codebeispiel 13.1: Schematische Verwendung des ForkJoin-Frameworks

Gestartet wird die Verarbeitung etwa wie folgt:

```
ForkJoinPool fjThreadPool = new ForkJoinPool();
SimpleTask  rootTask = new SimpleTask(...);
fjThreadPool.invoke( rootTask );
```

Der Threadpool muss hier nicht explizit beendet werden, da die Threads im `ForkJoinPool` die *Daemon*-Eigenschaft besitzen.

Codebeispiel 13.2 zeigt die Implementierung einer parallelen Array-Initialisierung. In der `compute`-Methode wird der zu initialisierende Bereich so lange halbiert, bis dessen Größe `THRESHOLD` erreicht hat (❶). Für die Teilbereiche werden jeweils neue Tasks erzeugt (❷).

```
class RandomInitTask extends RecursiveAction
{
  private final int THRESHOLD = 4;
  private final int[] array;
  private final int min;
  private final int max;
  private final int rdMax;

  RandomInitTask(int[] array, int min, int max, int rdMax)
  {
    this.array = array;
```

```
    this.min = min;
    this.max = max;
    this.rdMax = rdMax;
  }

  @Override
  protected void compute()
  {
    if( (max - min) <= THRESHOLD )                                    ❶
    {
      for(int i = min; i < max; i++)
      {
        array[i] = ThreadLocalRandom.current().nextInt(rdMax);
      }
    }
    else
    {
      int mid = min + (max-min)/2;                                    ❷
      RandomInitTask left = new RandomInitTask(array,min, mid, rdMax);
      RandomInitTask right = new RandomInitTask(array,mid, max, rdMax);
      invokeAll(left, right);
    }
  }
}
```

Codebeispiel 13.2: `RecursiveAction` für die Initialisierung eines `int`-Arrays

Das folgende Codebeispiel zeigt die Verwendung:

```
int[] array = new int[42];

ForkJoinPool fjPool = new ForkJoinPool();
RandomInitTask root = new RandomInitTask(array,0, array.length, 100);
fjPool.invoke(root);

System.out.println(Arrays.toString(array));
```

Die Initialisierung benutzt hierbei einen explizit erzeugten `ForkJoinPool`. Möchte man den ab Java 8 zur Verfügung gestellten internen Common-Pool (`ForkJoinPool.commonPool`) benutzen, so kann man direkt auf dem Task-Objekt `invoke` aufrufen:

```
int[] array = new int[42];

RandomInitTask root = new RandomInitTask(array,0, array.length, 100);
root.invoke();

System.out.println(Arrays.toString(array));
```

Klassische Anwendungen für `RecursiveAction` sind Divide-and-Conquer-Verfahren, bei denen kein Wert zurückgeliefert wird. Typische Beispiele sind Sortieralgorithmen, die *In-Place*-Ersetzungen durchführen. Die beiden bekanntesten sind Quick- und Mergesort.

Hinweis

`THRESHOLD`-Werte sollten sorgfältig gewählt werden. Sind sie zu klein, überwiegt der Overhead der Zerlegung und Zusammenführung und dies führt zu einer schlechteren Performance.

Praxistipp

Im Codebeispiel 13.2 werden beide `RandomInitTask`-Objekte durch `invokeAll(left, right)` asynchron gestartet. Es ist im Prinzip ausreichend, wenn nur ein Task asynchron ausgeführt wird und der andere direkt vom Aufrufer, wie im folgenden Codebeispiel:

```
RandomInitTask left = ...;
RandomInitTask right = ...;
left.fork();
right.compute();
left.join();
```

Hierbei ist zu beachten, dass das Starten des asynchronen Tasks (`fork`) vor dem direkten Ausführen des zweiten (`compute`) erfolgen muss. Außerdem darf man hier nicht vergessen, explizit auf das Ende des abgezweigten Tasks zu warten (`join`).

Die Variante mit `invokeAll` ist weniger fehleranfällig und sollte somit standardmäßig in der Praxis angewendet werden.

13.2.2 Einsatz von RecursiveTask

Soll durch die parallele Bearbeitung ein Ergebnis ermittelt werden, kann `RecursiveTask` eingesetzt werden. Man spricht in dem Zusammenhang auch oft von einer *Reduce*-Operation. Codebeispiel 13.3 zeigt einen `RecursiveTask` für die parallele Summation der Elemente eines `int`-Arrays.

```
class SumTask extends RecursiveTask<Integer>                          ❶
{
  private final int THRESHOLD = 4;

  private final int[] array;
  private final int min;
  private final int max;

  SumTask(int[] array, int min, int max)
  {
    this.array = array;
    this.min = min;
    this.max = max;
  }

  @Override
  protected Integer compute()                                         ❷
  {
    if( (max - min) <= THRESHOLD )
    {
      int count = 0;
      for(int i = min; i < max; i++)
      {
        count += array[i];
      }

      return count;                                                   ❸
    }
    else
    {
      int mid = min + (max-min)/2;
      SumTask left = new SumTask(array,min, mid );
      SumTask right = new SumTask(array,mid, max );
      invokeAll(left, right);

      return left.join() + right.join();                              ❹
    }
  }
}
```

Codebeispiel 13.3: `RecursiveTask` für die Summation eines `int`-Arrays

Der `RecursiveTask` wird mit dem Rückgabetyp parametrisiert (❶). Die `compute`-Methode erhält dadurch eine explizite Rückgabe (❷). In der *Work*-Phase wird der aktuelle Bereich aufsummiert und das Ergebnis zurückgegeben (❸). In der *Combine*-Phase werden die Ergebnisse der Teilberechnungen addiert (❹). Abbildung 13-4 zeigt den schematischen Ablauf.

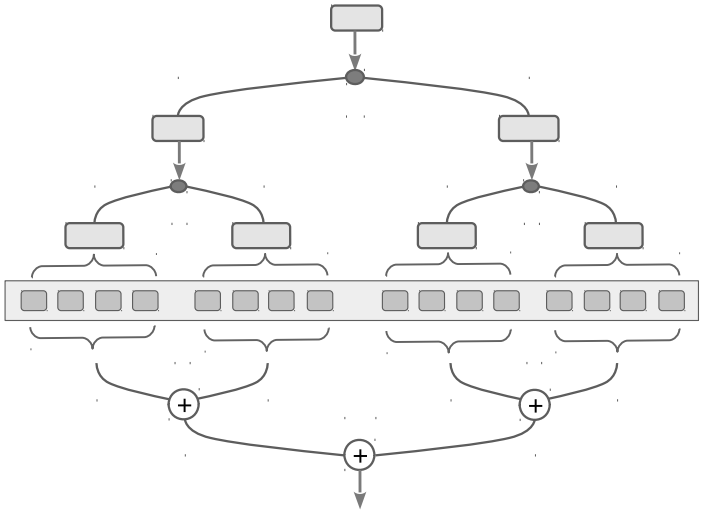

Abbildung 13-4: Parallele Summation eines Arrays

13.2.3 Einsatz von CountedCompleter

Die Klasse `CountedCompleter` hat gegenüber `RecursiveAction` bzw. `RecursiveTask` verschiedene Möglichkeiten, den rekursiven Ablauf zu steuern. Insbesondere können die Tasks manuell verwaltet werden. Die Klasse wird im Wesentlichen intern für die parallele Stream-Verarbeitung benutzt. Sie bietet unter anderem folgende Methoden an:

- `void addToPendingCount(int delta)`: Erhöht den internen Task-Zähler.
- `void tryComplete()`: Mit dieser Methode wird signalisiert, dass ein Task beendet ist und der interne Task-Zähler wird erniedrigt.
- `void quietlyCompleteRoot()`: Signalisiert dem Root-Task, dass ein Ergebnis vorliegt und dass er seine Blockierung aufheben kann.
- `E getRawResult()`: Die Methode stellt das Ergebnis der Berechnung bereit. Ist kein Ergebnis vorgesehen (die `CountedCompleter`-Klasse wurde mit `Void` parametrisiert), wird `Void` zurückgegeben.

Insbesondere kann mit diesen Methoden das Beenden (*completion*) einer parallelen Berechnung explizit kontrolliert werden.

Als Beispiel für den Einsatz von `CountedCompleter` betrachten wir eine Suche nach einem bestimmten Dateinamen. Sobald eine erste passende Datei gefunden wird, soll die Suche beendet und das Ergebnis ausgegeben werden. Das Suchmuster wird über einen regulären Ausdruck angegeben.

Codebeispiel 13.4 zeigt eine mögliche Implementierung. Die Klasse ist mit `Optional<File>` parametrisiert und überschreibt die beiden Metho-

den `compute` (❹) und `getRawResult` (❾). Das Suchergebnis wird in einer `AtomicReference` verwaltet (❶). In der `compute`-Methode wird der Inhalt eines Verzeichnisses ermittelt und durchlaufen (❹). Dabei wird immer zuerst geprüft, ob bereits ein Ergebnis vorliegt (❺). Falls ja, wird der Vorgang beendet. Trifft man auf ein Verzeichnis, wird ein neuer Task abgezweigt, wobei dem Framework dies explizit mit `addToPendingCount(1)` mitgeteilt wird (❻). Wurde eine Datei gefunden, wird geprüft, ob deren Name dem regulären Ausdruck entspricht. Bei Übereinstimmung wird noch zusätzlich geprüft, ob bereits schon etwas gefunden wurde (❼). Falls nichts vorliegt, wird das Ergebnis in der `AtomicReference` (❶) hinterlegt und mit `quietlyCompleteRoot` dem Framework signalisiert, dass die Suche erfolgreich ist. Die Blockierung des *Root*-Tasks wird hierdurch aufgehoben und der Aufrufer erhält das Ergebnis. Mit `tryComplete` wird dem Framework mitgeteilt, dass sich ein Task beendet hat (❽).

In diesem Beispiel werden zwei Konstruktoren verwendet. Der eine, als `public` deklariert, erhält als Parameter das Startverzeichnis für die Suche und den regulären Ausdruck (❸). Der `private`-Konstruktor, der von dem `public`-Konstruktor und in der `compute`-Methode benutzt wird, setzt über den ersten Parameter `parent` eine Referenz auf den Erzeuger-Task. Somit kann dann beim Aufruf von `quietlyCompleteRoot` intern das *Completed*-Signal zum *Root*-Task durchgereicht werden.

```
public class FindTask extends CountedCompleter<Optional<File>>
{
  private static final FileFilter fileFilter=new FileFilter()
  {
     public boolean accept(File f){
       return f.isDirectory()||f.isFile();
     }
   };

  private final File dir;
  private final String regex;
  private final AtomicReference<File> result;                    ❶

  public FindTask(File dir, String regex)                        ❷
  {
    this( null, dir, regex, new AtomicReference<File>( null) );
  }

  private FindTask(CountedCompleter<?> parent, File dir,         ❸
                   String regex,
                   AtomicReference<File> result)
  {
    super(parent);
    this.dir = dir;
    this.regex = regex;
    this.result = result;
  }
```

```
  @Override
  public void compute()                                                    ❹
  {
    File[] entries = dir.listFiles(fileFilter);

    if (entries != null )
    {
      for (File entry : entries)
      {
        if( result.get() != null )                                         ❺
          break;

        if (entry.isDirectory())
        {
          addToPendingCount(1);                                            ❻
          FindTask task =
                    new FindTask(this, entry, this.regex, result);
          task.fork();
        }
        else
        {
          String tmp = entry.getPath();
          if( tmp.matches(regex)
              && result.compareAndSet( null, entry ) )                     ❼
          {
            quietlyCompleteRoot();
            break;
          }
        }
      }
    }
    tryComplete();                                                         ❽
  }

  @Override
  public Optional<File> getRawResult()                                     ❾
  {
    File res = result.get();
    if( res == null )
      return Optional.empty();
    else
      return Optional.of(res);
  }
}
```

Codebeispiel 13.4: Ein Programm zur Dateisuche

Da eine Suche nicht immer einen Treffer liefert, wurde hier als Ergebnistyp ein `Optional<File>` benutzt, um `null` als Rückgabe zu vermeiden[1].

[1] `Optional` als Rückgabe einer Suche wird z. B. auch bei den entsprechenden `Stream`-Operationen verwendet (vgl. Kapitel 14).

Das folgende Codebeispiel zeigt die Verwendung der Klasse `FindTask`. Dabei wird direkt auf dem Task-Objekt die `invoke`-Methode aufgerufen und damit der CommonPool benutzt.

```
String search = ".*.java$";
File rootDir = new File("....");

FindTask root = new FindTask( rootDir, search );
root.invoke().ifPresent( System.out::println );

// Alternativer Aufruf
// root.invoke();
// root.join().ifPresent( System.out::println );

System.out.println("done");
```

13.3 Work-Stealing-Verfahren

In diesem Abschnitt wird das *Work-Stealing*-Verfahren an einem Beispiel näher erläutert. Das Verfahren ist das Rückgrat des ForkJoin-Frameworks. Würde man nämlich für jeden anfallenden Task einen neuen Thread starten, würde das zu einer exponentiell steigenden Anzahl von Threads führen.

Zum besseren Verständnis des Verfahrens betrachten wir die parallele Summation eines Arrays. Die abzuarbeitende Aufgabe wird hier, wie in Abbildung 13-5 gezeigt, in einzelne Tasks zerlegt. Der Root-Task entspricht t_0 und wird an das ForkJoin-Framework übergeben (vgl. hierzu auch Codebeispiel 13.3).

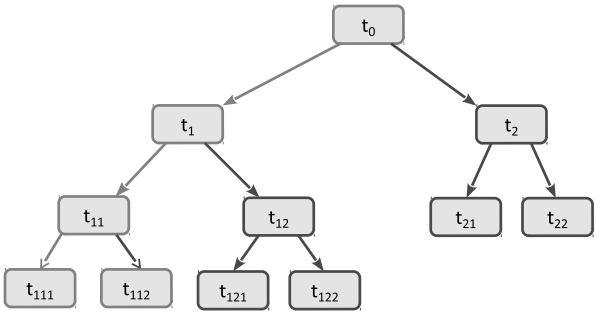

Abbildung 13-5: Parallele Summation eines Arrays

Unter der Annahme, dass zwei Threads zur Verfügung stehen, könnte dann die Aufgabe wie im folgenden Ablauf abgearbeitet werden. Dies ist nur eine von vielen Möglichkeiten, da die beiden Threads unabhängig voneinander

arbeiten. Der hier gewählte quasisynchrone Ablauf dient lediglich der besseren Veranschaulichung.

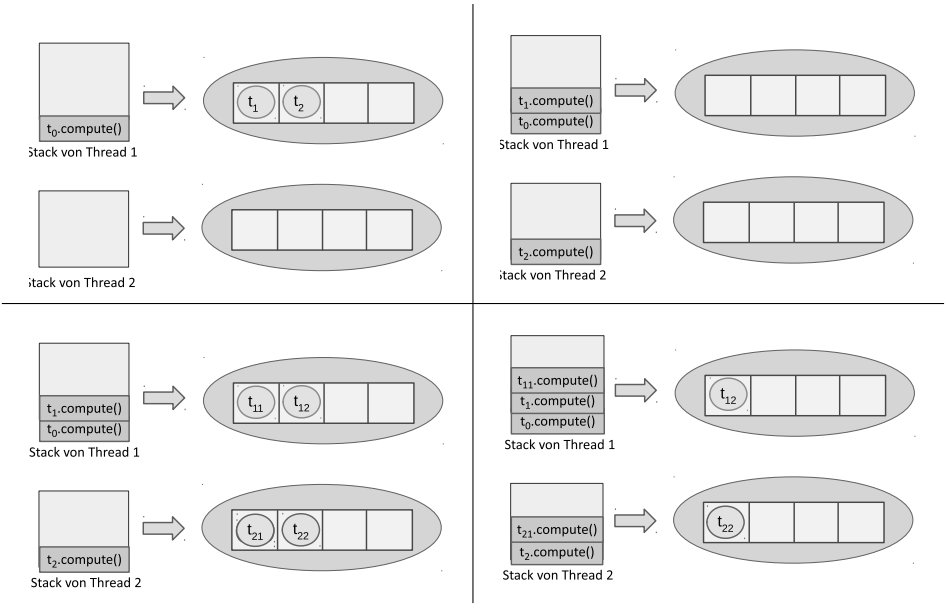

Abbildung 13-6: Der Beginn der Verarbeitung

Abbildung 13-6 bis 13-8 zeigen jeweils vereinfacht den Stack und die Workqueues der beiden Threads. Der zeitliche Ablauf ist zeilenweise jeweils von links nach rechts dargestellt.

Abbildung 13-6 zeigt die ersten Schritte. Durch die Übergabe des Tasks t_0 an das Framework, wird er in die Workqueue von `Thread 1` gestellt. Der Thread holt sich den Task und bearbeitet ihn. Das Problem wird in zwei neue Tasks zerteilt und in die Workqueue gestellt (`invokeAll(t1,t2)`, links oben). Thread 1 holt sich t_1 aus der Queue und bearbeitet ihn. Da Thread 2 nichts zu tun hat, holt er sich eine Arbeit vom Ende der Queue von Thread 1. In unserem Fall ist dies t_2 (rechts oben). Jetzt bearbeiten beide Threads ihre Aufgaben. Da sowohl t_1 als auch t_2 weiter zerlegt werden, werden die Workqueues mit neuen Task-Objekten (`invokeAll(t11,t12)` bzw. `invokeAll(t21,t22)`) gefüllt (vgl. Abb. links unten). Danach holt sich Thread 1 hier t_{11} und Thread 2 t_{21} (rechts unten).

Abbildung 13-7 zeigt die folgenden Verarbeitungsschritte und Abbildung 13-8 illustriert die Endphase der Verarbeitung.

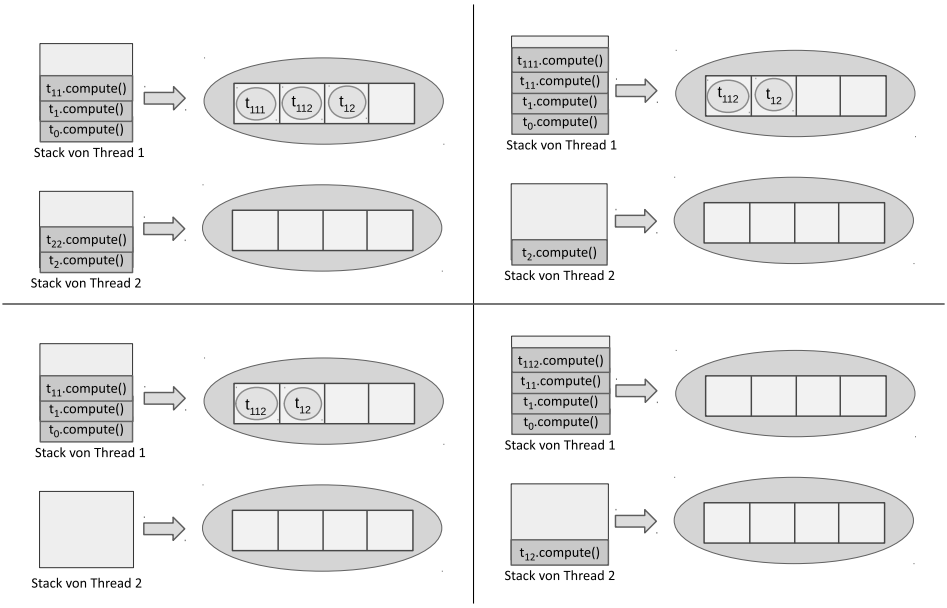

Abbildung 13-7: Weitere Schritte der Verarbeitung

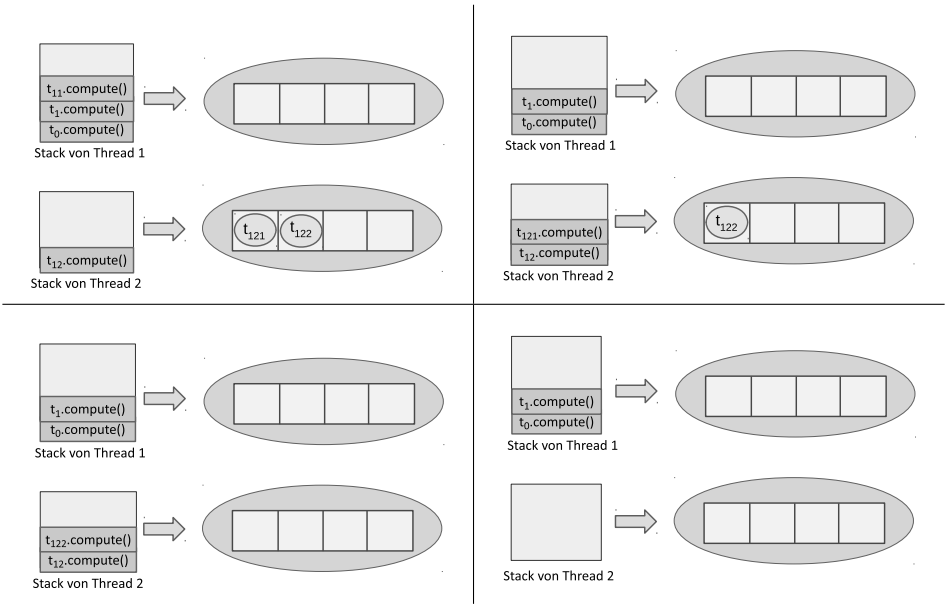

Abbildung 13-8: Endphase der Verarbeitung

13.4 Zusammenfassung

Mit dem ForkJoin-Framework steht ein leistungsfähiger Mechanismus zur Verfügung, mit dem Berechnungen nach dem Divide-and-Conquer-Verfahren parallel abgearbeitet werden können. Das Framework erweitert das Prinzip des `Future`-Patterns und stellt die Klassen `RecursiveAction`, `RecursiveTask` und `CountedCompleter` zur Verfügung, die entsprechend der zu parallelisierenden Aufgabe abgeleitet werden können.

Der mit dem ForkJoin-Framework eingeführte `ForkJoinPool` unterstützt das Work-Stealing-Verfahren, sodass mit einer kleinen Menge von Threads auch tiefe Task-Hierarchien und somit eine sehr große Anzahl an Tasks bewältigt werden können.