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].

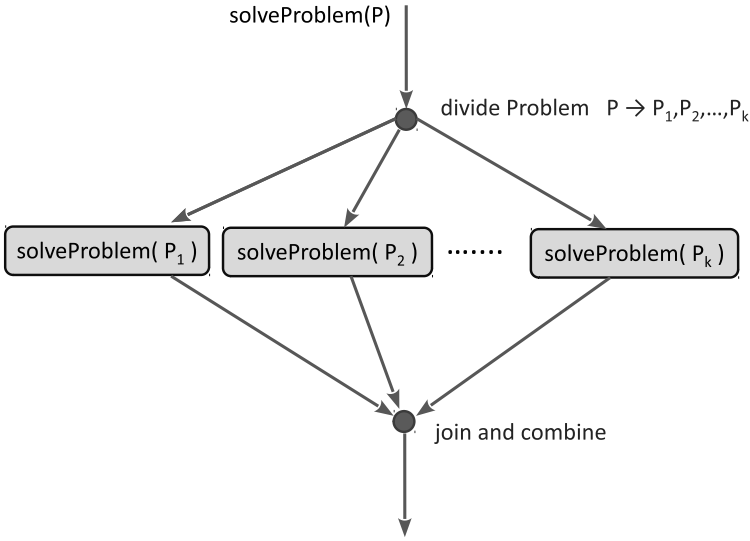

Abbildung 13-1: Der Kontrollfluss des ForkJoin-Patterns

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

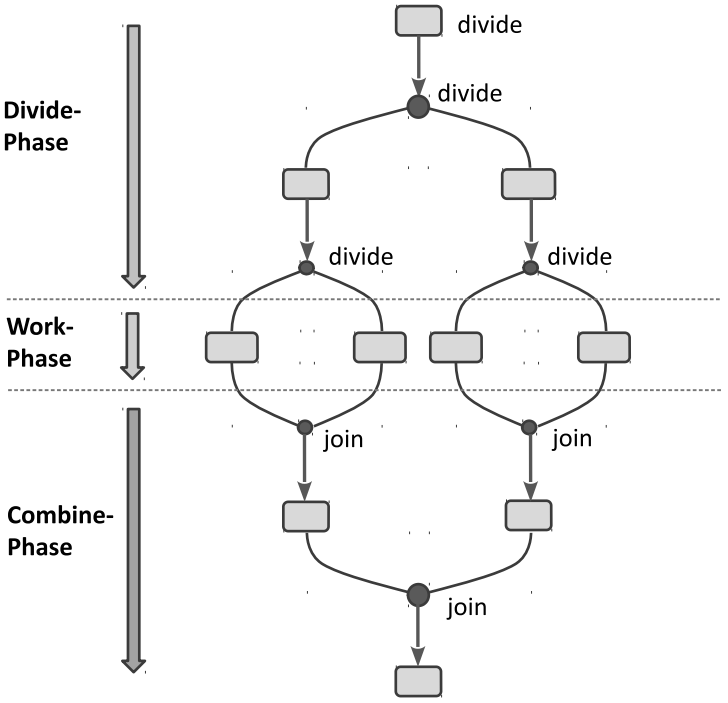

Abbildung 13-2: Rekursive Verwendung des ForkJoin-Patterns

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.

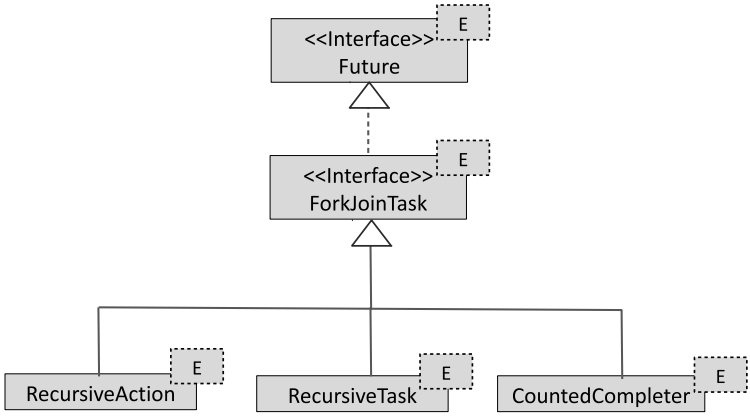

Abbildung 13-3: Hierarchie der `Task`-Klassen

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.

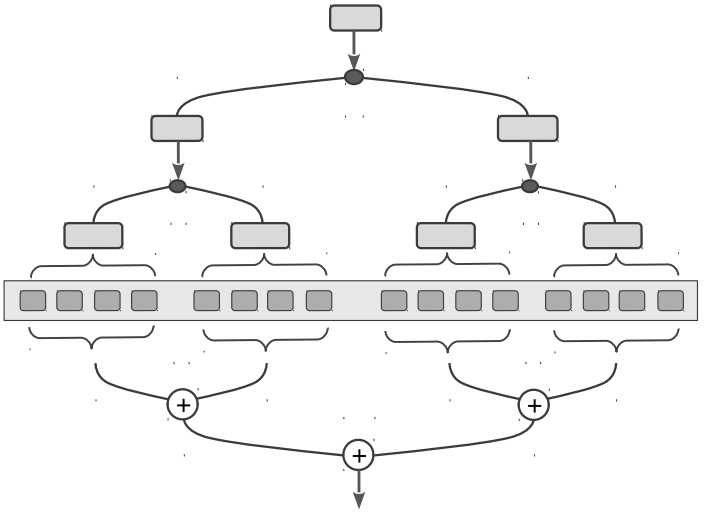

Abbildung 13-4: Parallele Summation eines Arrays

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).

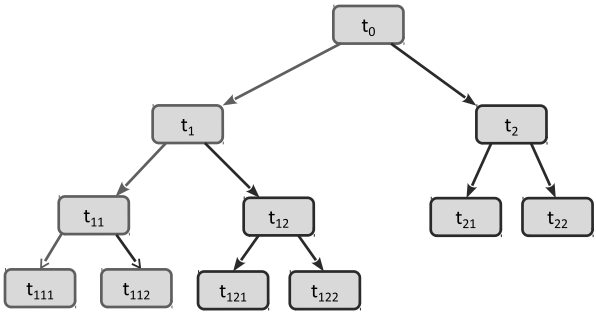

Abbildung 13-5: Parallele Summation eines Arrays

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.

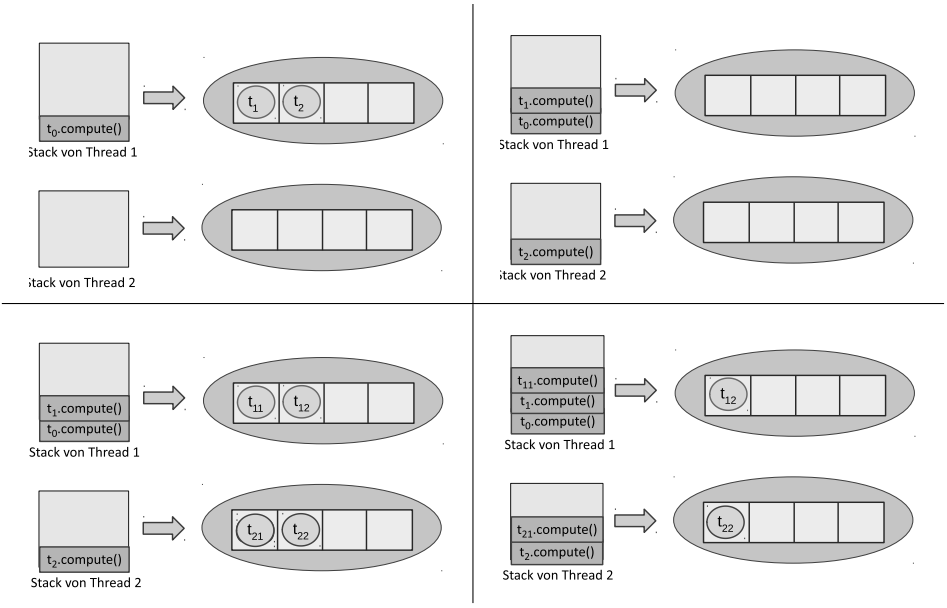

Abbildung 13-6: Der Beginn der Verarbeitung

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.

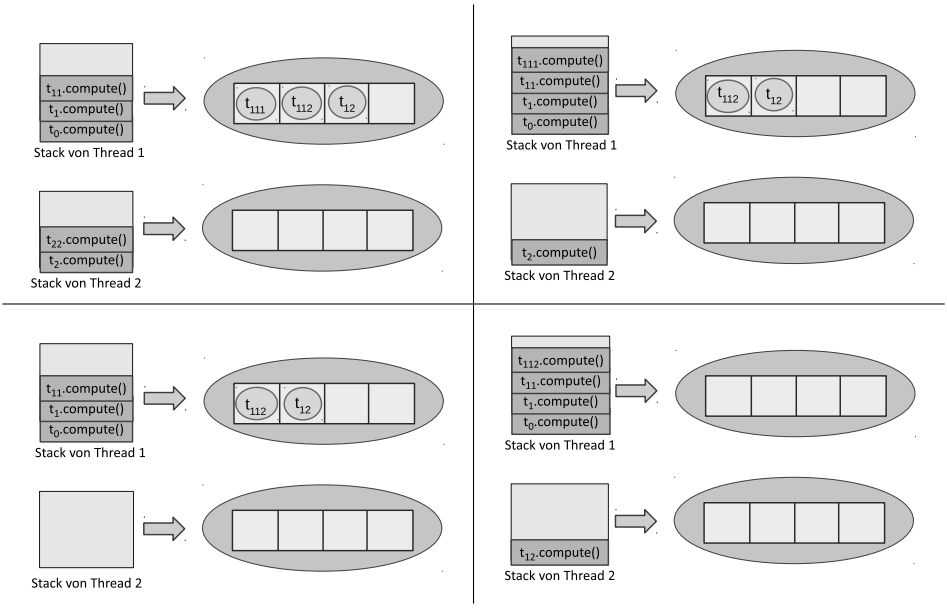

Abbildung 13-7: Weitere Schritte der Verarbeitung

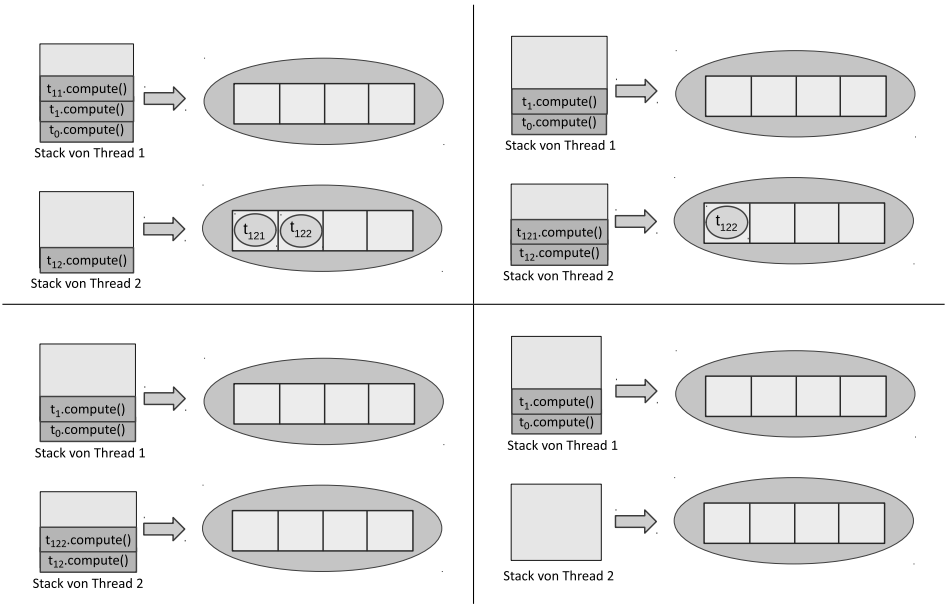

Abbildung 13-8: Endphase der Verarbeitung

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.