

Algorithmen & Datenstrukturen

Parallelisierungsframeworks

When you come to a fork in the road.

Take it. – Yogi Berra

Roger Diehl



Inhalt

- Parallelisierungsframeworks
- Das Fork-Join Konzept
- Das Fork-Join Programmiermodell
- Einsatz von **RecursiveAction**
- Einsatz von **RecursiveTask**
- Einsatz von **CountedCompleter**
- Zusammenfassung

Lernziele

- Sie kennen das Grundprinzip des Fork-Join Konzepts.
- Sie kennen das ForkJoin-Framework von Java.
- Sie wissen welche abstrakte Klasse des ForkJoin-Framework Sie ableiten müssen,
 - für Aufgaben ohne Rückgabe,
 - für Aufgaben mit Rückgabe,
 - für Aufgaben mit speziellem Warten auf das Ende der parallelen Bearbeitung.
- Sie können den mit dem ForkJoin-Framework eingeführte **ForkJoinPool** verwenden.
- Sie kennen das Work-Stealing-Verfahren des **ForkJoinPool**.

Parallelisierungsframeworks

- Fork-Join-Framework
 - Es wird für die Parallelisierung von rekursiven Divide-and-Conquer-Algorithmen eingesetzt.
 - Es verwendet einen Threadpool mit Work-Stealing-Verfahren.
- Parallele Array- und Stream-Verarbeitung
 - Mit Java 8 wurden sowohl für Arrays als auch für Collections parallele Verarbeitungsmöglichkeiten eingeführt.
 - Streams sind Abstraktionen, welche die sequentielle oder parallele Ausführung von Operationen auf Elemente einer Sequenz unterstützen.
- Completable Future
 - Mit Java 8 gibt es die Erweiterung des Future-Patterns.
 - Damit kann man Tasks mit asynchronen Ergebnisse handhaben.

Das Fork-Join Konzept

Grundprinzip des Fork-Join Konzepts

- Aufteilen (Fork) und Vereinen (Join)
- **Fork** wird von einem (übergeordneten) Thread aufgerufen, um einen neuen (Kind-) Thread zu erstellen.
 - Der «Elternteil» fährt nach dem Fork-Vorgang weiter.
 - Das «Kind» beginnt die Operation getrennt vom «Elternteil».
 - Fork schafft Nebenläufigkeit.
- **Join** wird von beiden, «Elternteil» und «Kind», aufgerufen.
 - «Kinder» rufen Join auf, wenn die Operation zu Ende ist (implizit beim Beenden des Kind-Thread).
 - Der «Elternteil» wartet, bis das «Kind» die Operation beendet hat (joins) und fährt danach fort.
 - Join reduziert die Nebenläufigkeit, weil das «Kind» beendet wird.

Bedeutung von Fork-Join für die Nebenläufigkeit

- Abhängigkeitsregeln von Fork-Join
 - Der «Elternteil» muss auf seine «Kinder» warten.
 - Aufgeteilte «Kinder» vom gleichen «Elternteil» können in beliebiger Reihenfolge wieder zusammen kommen.
 - Ein «Kind» kann nicht mit seinem «Eltern» zusammenkommen, bis es mit allen «Kindern» verbunden ist.
- Fork-Join ist ein Kontrollmechanismus für die Nebenläufigkeit.
 - Fork erhöht die Nebenläufigkeit.
 - Join reduziert die Nebenläufigkeit.

Das Fork-Join-Pattern

- Ein typischer Divide-and-Conquer-Algorithmus

```
function DivideAndConquer(Problem P)
```

```
  if P:size < THRESHOLD then
```

```
    solve P sequentially
```

```
  else
```

```
    -> Divide P in k subproblems  $P_1; P_2; \dots; P_k$ 
```

```
    -> Fork to conquer each subproblem in parallel
```

```
    fork DivideAndConquer( $P_1$ )
```

```
    fork DivideAndConquer( $P_2$ )
```

```
    fork ...
```

```
    fork DivideAndConquer( $P_k$ )
```

```
    join
```

```
    -> Combine subsolutions into final solution
```

```
  end if
```

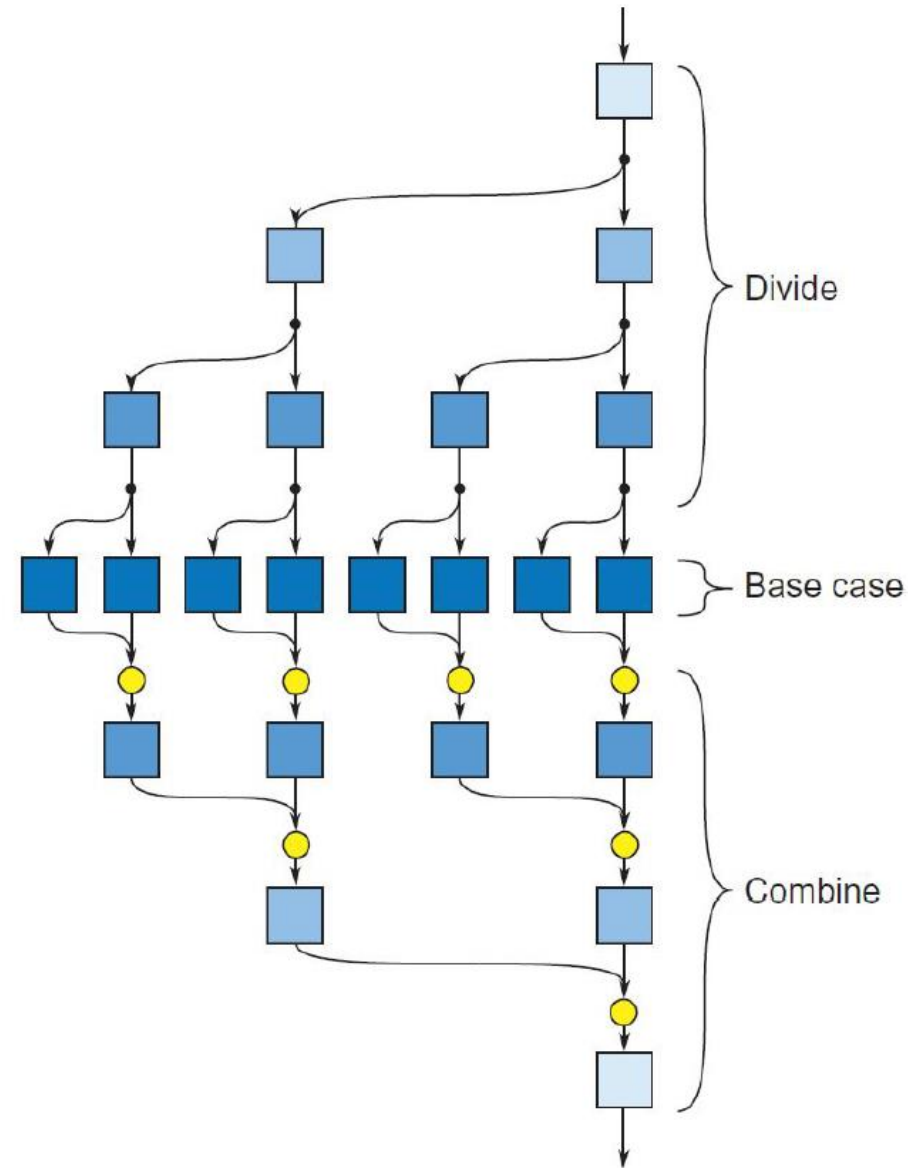
```
end function
```

Pseudocode

Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

Kontrollfluss des Fork-Join-Pattern

- Beim Fork-Join-Pattern wird der Kontrollfluss an einer dedizierten Stelle in mehrere nebenläufige Flüsse aufgeteilt (fork), die an einer späteren Stelle alle wieder vereint (join) werden.
- Die Vereinigung entspricht einem Synchronisationspunkt.
- Wenn alle Teilaufgaben erledigt sind, wird das Programm danach fortgesetzt.

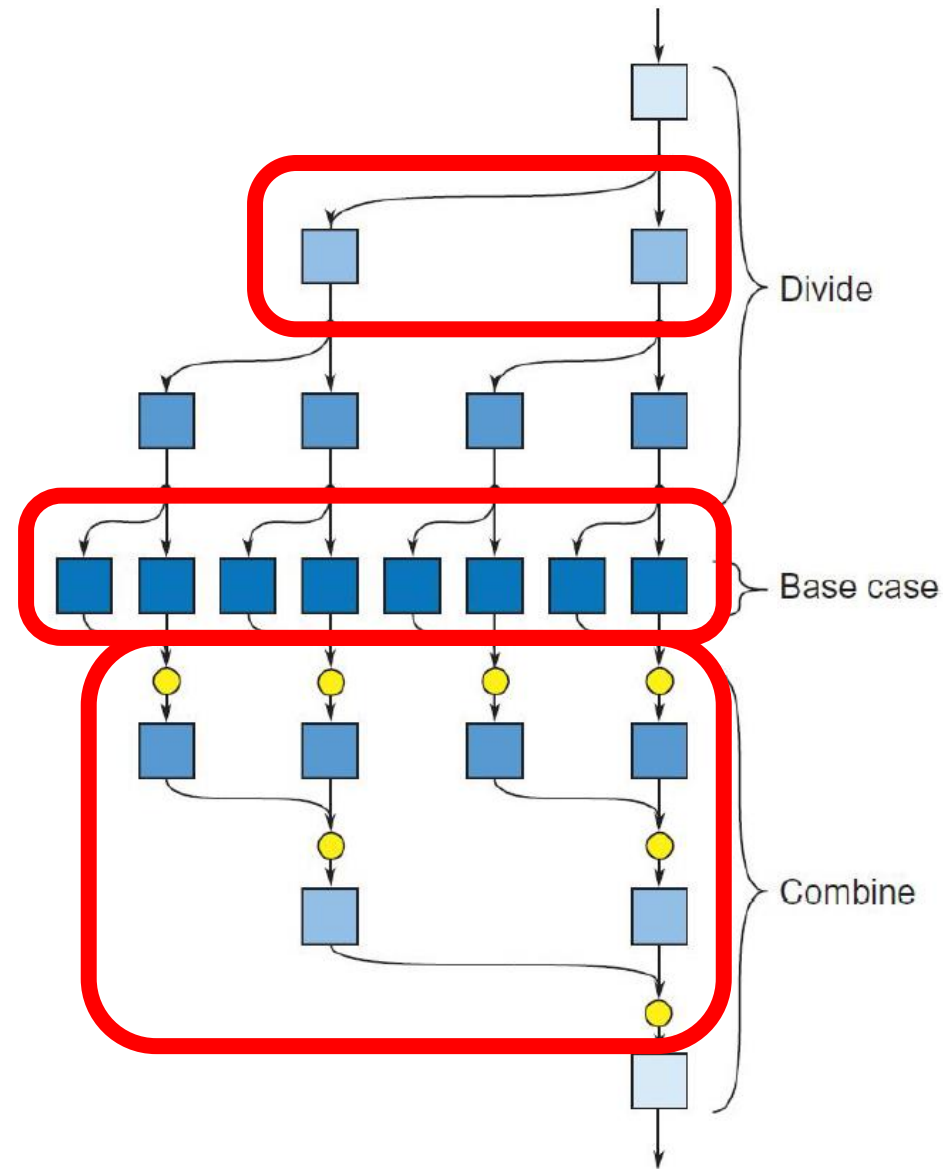


Fork-Join-Pattern Faktoren

Aufteilung Sub-Problems
Divider: $K = 2$

Parallelisierungsgrad:
 $K^N = 2^3 = 8$

Vereinigung Sub-Solutions
Combine Level: $N = 3$



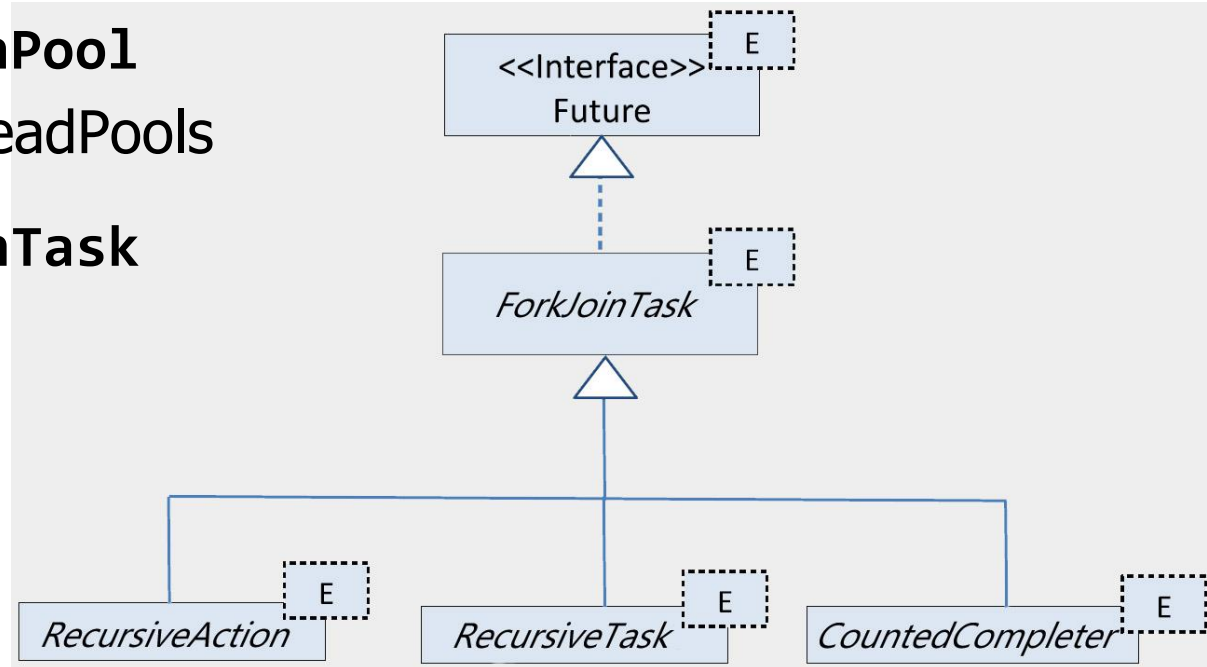
Folgerungen zu Fork-Join-Pattern

- Die Auswahl der Grösse des Basisfalles ist kritisch.
- Rekursion muss tief genug gehen für möglichst viel Parallelität.
- Allerdings zu tief – und die Granularität der Teilaufgaben wird durch das Thread-Scheduling dominiert.
- Mit K Aufteilungen und N Vereinigungslevels, kann ein Parallelisierungsgrad von bis zu K^N erreicht werden.

Das Fork-Join Programmiermodell

Komponenten des ForkJoin-Frameworks

- Threadpool **ForkJoinPool**
 - Siehe N22_IP_ThreadPools
- Abstraktion **ForkJoinTask**
 - vom Typ **Future**
- Spezialisierte Klassen
 - **RecursiveAction**
 - **RecursiveTask**
 - **CountedCompleter**



Quelle: Hettel/Tran - Nebenläufige Programmierung mit Java

- Eine konkrete Aufgabe, muss abgeleitet werden.
 - Für Aufgaben ohne Rückgabe von **RecursiveAction**.
 - Für Aufgaben mit Rückgabe von **RecursiveTask**.
 - Für Aufgaben mit speziellem Warten auf das Ende der Sub-Tasks von **CountedCompleter** (seit Java 1.8).

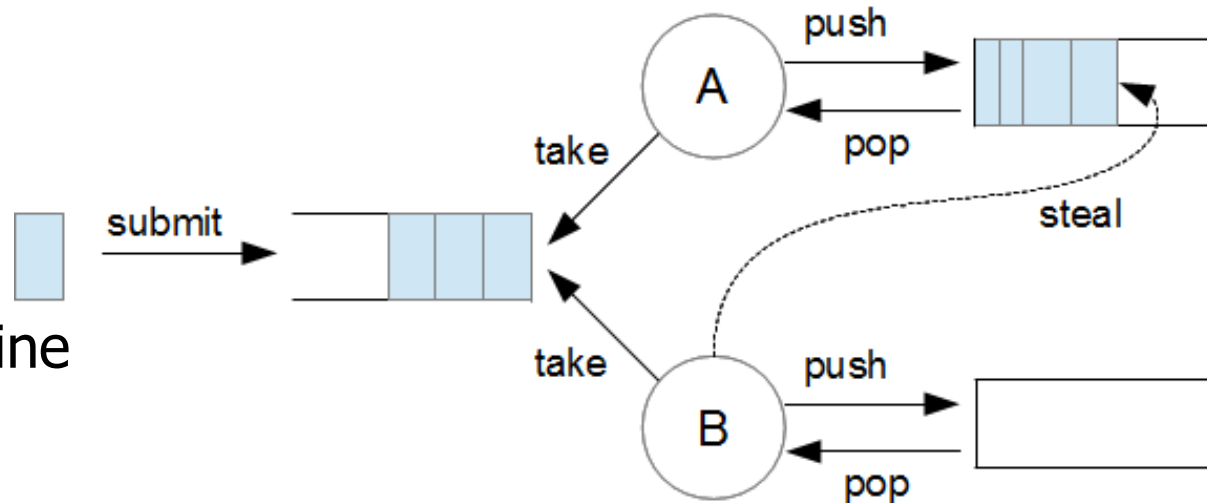
Wichtige Methoden des ForkJoinPool Threadpools

- **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.

Work-Stealing-Verfahren

- 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.

- Bei diesem Verfahren besitzt jeder Thread eine eigene Task-Queue, aus der er seine Aufträge holt bzw. Aufträge hineinstellt.



Quelle: heise.de/developer/artikel/Das-Fork-Join-Framework

- **Der Trick:** Ist die Queue des Threads leer, holt er sich vom Ende anderer Task-Queues Aufgaben und bearbeitet diese.
 - Double-Ended-Queues (**Deque**) kommen hier zum Einsatz.

Abstraktion ForkJoinTask

- Die von den Tasks zu implementierende Methode ist **compute**, in der die Aufteilung der Aufgabe und die Verzweigung in die Teilaufgaben durchgeführt wird.
- Für die Verzweigung stehen die Methoden **fork** und **invoke** zur Verfügung.
 - Mit **fork** wird die asynchrone, nicht blockierende Ausführung des Tasks gestartet.
 - Mit **invoke** wird die synchrone, blockierende Ausführung des Tasks gestartet.
- Mit **join** kann das Ergebnis der Verarbeitung abgeholt werden.
- Die von Future implementierte Methode **get** verhält sich wie **join**, wirft aber im Fehlerfall eine **InterruptedException** oder **ExecutionException**.

Übersicht der ForkJoin-Framework Methoden

- Die Methoden **execute**, **invoke**, **submit** dienen als Startpunkte für Divide-and-Conquer-Algorithmen.
- Die Methoden **fork** und **invoke** werden 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	<code>execute(ForkJoinTask)</code>	<code>ForkJoinTask.fork()</code>
Synchrone Ausführung	<code>invoke(ForkJoinTask)</code>	<code>ForkJoinTask.invoke()</code>
Asynchrone Ausführung mit Rückgabewert über Future-Objekt	<code>submit(ForkJoinTask)</code>	<code>ForkJoinTask.fork()</code>

Schematische Verwendung des ForkJoin-Frameworks

- Für den Einsatz des ForkJoin-Frameworks gibt es immer ein ähnliches Code-Template.

```
public final class SimpleTask extends RecursiveAction {  
    // Attribute  
    // Konstruktoren  
    @Override  
    protected void compute() {  
        if (n <= SEQUENTIAL_THRESHOLD) {  
            // Sequenzieller Algorithmus  
        } else {  
            // Definition von mehreren 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);  
        }  
    }  
}
```

Codeskizze

 **invokeAll** blockiert und kehrt erst zurück, wenn alle Teilaufgaben beendet sind.

Start der Verarbeitung beim ForkJoin-Framework

- Gestartet wird die Verarbeitung wie folgt:

```
final ForkJoinPool forkJoinPool = new ForkJoinPool();  
final SimpleTask rootTask = new SimpleTask();  
forkJoinPool.invoke(rootTask);
```

- Der Threadpool muss hier nicht explizit beendet werden, da die Threads im **ForkJoinPool** die Daemon-Eigenschaft besitzen.
- Für viele Anwendungen genügt der Common Pool (seit Java 1.8), der zum Einsatz kommt, beim Aufruf von **invoke** am Task-Objekt.

```
final SimpleTask rootTask = new SimpleTask();  
rootTask.invoke();
```

- Der Vorteil hier ist, dass Ressourcen geschont werden, in dem Threads während der Nichtbenutzung langsam zurückgefahren und bei der späteren Verwendung wiederhergestellt werden.

Einsatz von RecursiveAction

Motivation – Parallelisierung von Mergesort

- siehe AD Input → **A21_IP_HöhereSortieralgorithmen**
- Mergesort verwendet das Lösungsprinzip «Teile und Herrsche». Es kann rekursiv beschrieben werden:
 - **Rekursionsbasis:**
 - Eine zu sortierende Folge von **einem** Element ist sortiert.
 - **Rekursionsvorschrift:**
 1. Die zu sortierende Folge von **mehreren** Elementen in zwei Hälften teilen.
 2. Sortieren der linken und der rechten Hälfte.
 3. Zusammenfügen der beiden sortierten Hälften zur sortierten Folge, und zwar mit dem «Reissverschlussverfahren» bzw. durch «Mischen» (to merge).

Praxistipp

- Das gezeigte Lösungsprinzip hat einige Nachteile und sollte so nicht implementiert werden.
- Es ist nicht ratsam, immer bis zu Teilbereichen der Länge **EINS** zu splitten.
 - Die Granularität der Teilbereiche könnte dann durch das Thread-Scheduling dominiert werden.
- Man sollte die Split-Phase bis zu einer gewissen Grösse durchführen und dann z.B. auf eine sequenzielle Verfahren zurückgreifen.

Initialisierung

- Die Sortieraufgabe muss von **RecursiveAction** abgeleitet werden.

```
public final class SortTask extends RecursiveAction
```

- Es werden Attribute für das Array sowie die Minimum- und Maximum-Werte für den zu sortierenden Bereich benötigt.

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

- Zudem legt die Schwelle **THRESHOLD** die Phase der sequentiellen Sortierung fest.

```
private static final int THRESHOLD = 5;
```

- Es gibt zwei Konstruktoren.
 - Einen öffentlichen, der das Array zum Sortieren entgegen nimmt.
 - Einen privaten, der das Array und den zu sortierenden Bereich festlegt, bzw. initialisiert.

Rekursionsbasis

- In der Methode **compute** findet Aufteilung der Aufgabe und die Verzweigung in die Teil-Aufgaben statt.
- Die Schwelle legt den sequentiell auszuführenden Teil fest.

```
@Override
protected void compute() {
    if (max - min < THRESHOLD) {
        sortSequentially(min, max);
    } else {
        // Definition von Sub-Tasks...
    }
}
```

- Für die sequentielle Sortierung wird die **Array.sort** Methode zu Hilfe genommen.

```
private void sortSequentially(final int min, final int max) {
    Arrays.sort(array, min, max);
}
```


Rekursionsvorschrift

1. Die zu sortierende Folge von mehreren Elementen in zwei Hälften teilen. Dazu wird die Mitte (mid) berechnet.
2. Sortieren der linken und der rechten Hälfte.
3. Zusammenfügen der beiden sortierten Hälften.

```
@Override
protected void compute() {
    if (max - min < THRESHOLD) {
        sortSequentially(min, max);
    } else {
        final int mid = min + (max - min) / 2;
        invokeAll(
            new SortTask(array, min, mid),
            new SortTask(array, mid, max));
        merge(min, mid, max);
    }
}
```

1. halbieren
2. sortiere
 • links
 • rechts
3. mergen

← **invokeAll**
blockiert.

Mischen mit dem Reissverschlussverfahren

```
private void merge(final int min, int mid, int max) {  
    int[] buf = Arrays.copyOfRange(array, min, mid);  
    int i = 0;  
    int j = min;  
    int k = mid;  
    while (i < buf.length) {  
        if (k == max || buf[i] < array[k]) {  
            array[j] = buf[i];  
            i++;  
        } else {  
            array[j] = array[k];  
            k++;  
        }  
        j++;  
    }  
}
```

Arrays.copyOfRange kopiert die Array Elemente von **min** bis **mid-1** ins Hilfsarray.

Vergleiche die beiden Elemente **i** und **k** der sortierten Hälften vom Hilfsarray und Array.

Kopiere das Kleinere und füge es dem Array an der Stelle **j** hinzu. Falls die Elemente gleich sind, so kopiere das Element vom Array.

Falls **k == max** ist, sind noch Elemente im Hilfsarray und müssen ins Array zurückkopiert werden.

Start der Sortierung

- Zur Sortierung wird benötigt:
 - Ein Integer Array
 - Ein Join-Fork-Threadpool
 - Einen Task zur Zufalls-Initialisierung des Integer Arrays (siehe Code*)
 - Einen Task zur Sortierung
 - Die beiden Tasks werden jeweils mit **invoke** dem Join-Fork-Threadpool übergeben. Die Methode **invoke** ist blockierend.

```
final int[] array = new int[42];
final ForkJoinPool pool = new ForkJoinPool();
// Initialisierung des Array...
final RandomInitTask initTask = new RandomInitTask(array, 100);
pool.invoke(initTask);
LOG.info(Arrays.toString(array));
// Sortierung des Array...
final SortTask sortTask = new SortTask(array);
pool.invoke(sortTask);
LOG.info(Arrays.toString(array));
```

Einsatz von RecursiveTask

Motivation – Parallelisierung der Quersummenberechnung

- Um sicher zu gehen, dass das vorgestellte Sortierverfahren korrekt arbeitet, ist die Quersumme des Arrays eine Prüfmöglichkeit.
- Quersummen von Arrays lassen sich auch nach dem Lösungsprinzip «Teile und Herrsche» berechnen. Es kann rekursiv beschrieben werden:
 - **Rekursionsbasis:**
 - Zwei Elemente addieren.
 - **Rekursionsvorschrift:**
 1. Teilen der zu berechnenden Folge von Elementen in zwei Hälften.
 2. Berechnen der linken und der rechten Hälfte.
 3. Addieren der beiden Resultate.

Zurückgeben von Ergebnissen bei der Rekursion

- Wenn durch die parallele Bearbeitung ein Ergebnis ermittelt wird, bietet sich die Verwendung von **RecursiveTask** an.
- Dieser Vorgang wird oft auch als Reduce-Operation bezeichnet.
- Der **RecursiveTask** wird mit dem Rückgabetyt parametrisiert.
- Die **compute** Methode erhält dadurch eine explizite Rückgabe.
- In der Work-Phase (Rekursionsbasis) wird der aktuelle Bereich berechnet und das Ergebnis zurückgegeben.
- In der Combine-Phase (Rekursionsvorschrift) werden die Ergebnisse der Teilberechnungen zusammengeführt, bzw. berechnet.

Initialisierung

- Die Berechnung muss von **RecursiveTask** abgeleitet und mit dem Rückgabebetyp **Integer** für die berechnete Summe parametrisiert werden.

```
public final class SumTask extends RecursiveTask<Integer>
```

- Es werden die gleichen Attribute wie beim Sortieren benötigt. Minimum- und Maximum-Werte für den zu berechnenden Bereich.

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

- Zudem legt die Schwelle **THRESHOLD** die Phase der sequentiellen Summen-Berechnung fest (wie beim Sortieren).

```
private static final int THRESHOLD = 4;
```

- Es gibt zwei Konstruktoren (wie beim Sortieren).

Rekursionsbasis

- In der Methode **compute** findet Aufteilung der Aufgabe und die Verzweigung in die Teil-Aufgaben statt (wie beim Sortieren).
- Die Schwelle legt den sequentiell auszuführenden Teil fest. Auch hier empfiehlt es sich, die Schwelle nicht zu tief zu legen.
- Für die sequentielle Sortierung wird eine **for**-Iteration zu Hilfe genommen.

```
@Override
protected Integer compute() {
    int sum = 0;
    if ((max - min) <= THRESHOLD) {
        for (int i = min; i < max; i++) {
            sum += array[i];
        }
    } else {
        // Definition von Sub-Tasks...
    }
}
```


Rekursionsvorschrift

1. Teilen der zu berechnenden Folge von Elementen in zwei Hälften.
Dazu wird die Mitte (*mid*) berechnet.
2. Berechne die linke und die rechte Hälfte.
3. Addiere die beiden Resultate zusammen.

```
@Override
protected Integer compute() {
    int sum = 0;
    if ((max - min) <= THRESHOLD) {
        // Definition sequentiellen Berechnung...
    } else {
        final int mid = min + (max - min) / 2;
        final SumTask taskLeft = new SumTask(array, min, mid);
        taskLeft.fork();
        final SumTask taskRight = new SumTask(array, mid, max);
        sum = taskRight.invoke() + taskLeft.join();
    }
    return sum;
}
```

taskLeft asynchron starten

taskRight synchron starten – auf Rückgabewert warten

join blockiert – warten auf Rückgabewert von taskLeft

Einsatz von CountedCompleter

Motivation – Parallelisierung der Suche

- Elemente in nicht-sortierten Arrays lassen sich ebenfalls nach dem Lösungsprinzip «Teile und Herrsche» suchen. Es kann rekursiv beschrieben werden:
 - **Rekursionsbasis:**
 - Such-Resultat für ein Element zurück geben.
 - **Rekursionsvorschrift:**
 1. Teilen der Folge von Elementen in zwei Hälften.
 2. Suchen in der linken und in der rechten Hälfte.
 3. Such-Resultate zurück geben.
- **Halt!** – Hier wird zu viel gesucht.
 - Sobald ein Resultat gefunden wurde, kann die Suche beendet werden.

Erweiterungen von CountedCompleter

- **CountedCompleter** hat verschiedene Möglichkeiten, den rekursiven Ablauf zu steuern. Insbesondere können die Tasks mit Hilfe eines Zählers manuell verwaltet werden.
- **CountedCompleter** bietet Methoden für das explizite Beenden (completion) einer parallelen Berechnung an.
 - **void addToPendingCount(int delta)** – Erhöht den internen Task-Zähler.
 - **void tryComplete()** – Mit dieser Methode wird signalisiert, dass ein Task beendet ist.
 - **void quietlyCompleteRoot()** – Signalisiert dem Root-Task, dass ein Ergebnis vorliegt und er seine Blockierung aufheben kann.
 - **E getResult()** – Die Methode stellt das Ergebnis der Berechnung bereit. Ist kein Ergebnis vorgesehen, wird **Void** zurückgegeben.

Initialisierung

- Die Suche muss von **CountedCompleter** abgeleitet und mit dem Rückgabebetyp **Integer** für den Index des gefundenen Elements parametrisiert werden.

```
public final class SearchTask extends CountedCompleter<Integer>
```

- Ein Attribut **key** wird für den Suchschlüssel benötigt, sowie die gleichen Attribute wie bei der Summenberechnung und Sortierung.

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

- Die Schwelle **THRESHOLD** legt die Phase der sequentiellen Suche fest (wie bei der Summenberechnung und Sortierung).

```
private static final int THRESHOLD = 5;
```

Completed-Signal

- Es werden zwei Konstruktoren verwendet.
- Der **public** deklarierte, erhält den regulären Ausdruck als Parameter und zu durchsuchende Array.

```
public SearchTask(final int key, final int[] array)
```

- Der **private** Konstruktor, welcher im **public** Konstruktor und in der **compute**-Methode benutzt wird, setzt über den ersten Parameter **parent** eine Referenz auf den Erzeuger-Task.

```
private SearchTask(  
    final CountedCompleter<?> parent, final int key,  
    final int[] array, final int min, final int max,  
    final AtomicInteger result) {  
    super(parent);  
    // weitere Inits...
```

- Somit kann beim Aufruf von **quietlyCompleteRoot** intern das Completed-Signal zum Root-Task durchgereicht werden.

Suchergebnis

- Das Suchergebnis wird in einem **AtomicInteger** Attribut verwaltet, weil das Suchergebnis nebenläufig ermittelt wird.

```
private final AtomicInteger result;
```

- Das Suchergebnis wird im **public** Konstruktor mit -1 (kein Element gefunden) initialisiert.

```
public SearchTask(final int key, final int[] array) {  
    this(null, key, array, 0, array.length,  
        new AtomicInteger(-1));  
}
```

- Die Methode **getRawResult** stellt das Ergebnis der Suche bereit.

```
@Override  
public Integer getRawResult() {  
    return result.get();  
}
```

Rekursionsbasis

- In der Methode **compute** findet Aufteilung der Aufgabe und die Verzweigung in die Teil-Aufgaben statt (wie schon gezeigt).
- Für die sequentielle Suche wird eine **for**-Iteration zu Hilfe genommen.

```
@Override
protected void compute() {
    if ((max - min) <= THRESHOLD) {
        for (int i = min; i < max; i++) {
            if (array[i] == key && result.compareAndSet(-1, i)) {
                this.quietlyCompleteRoot();
                break;
            }
        }
    } else {
        // Definition von Sub-Tasks...
    }
}
```

Suchergebnis beim Index *i* gefunden.

Suchergebnis beim Resultat setzen.

Der Anfangswert von **result** ist -1.

Das Completed-Signal zum Root-Task durchreichen.

Rekursionsvorschrift

1. Teilen der Folge in zwei Hälften.
2. Suche in der linken und der rechten Hälfte starten.

```
@Override
protected void compute() {
    if ((max - min) <= THRESHOLD) {
        // Definition sequentiellen Berechnung.
    } else {
        final int mid = min + (max - min) / 2;
        this.addToPendingCount(2);
        final SearchTask taskLeft =
            new SearchTask(this, key, array, min, mid, result);
        taskLeft.fork();
        final SearchTask taskRight =
            new SearchTask(this, key, array, mid, max, result);
        taskRight.fork();
    }
    this.tryComplete();
}
```

Dem Framework
explizit die neuen
Tasks mitteilen
und zwar VOR
dem Task-Start.

taskLeft und **taskRight**
asynchron starten.

Dem Framework mitteilen,
dass ein Task beendet hat.

Optimierungen

- Siehe Java 8 Dokumentation über die Klasse **CountedCompleter**
<http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountedCompleter.html>
- Statt zwei Tasks starten und dann nichts mehr tun – nur einer mit **fork** starten und den zweiten direkt mit **compute** starten und auf das Resultat warten.
- Oft müssen gar nicht zwei Tasks gestartet werden – eine Aufgabe kann iterativ in Sub-Tasks aufgeteilt werden. Jeder Sub-Task wird gezählt und mit **fork** gestartet.
 - Für die parallele Stream-Verarbeitung (Java 8) bietet sich diese Art der Verarbeitung an.
 - **CountedCompleter** wurde im Wesentlichen dafür entwickelt.

Zusammenfassung

- Mit dem ForkJoin-Framework steht ein leistungsfähiger Mechanismus zur Verfügung, mit dem Algorithmen nach dem Divide-and-Conquer-Verfahren parallel abgearbeitet werden können.
- Das Framework erweitert das Prinzip des **Future**-Patterns und stellt die abstrakten 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.

Fragen?