12.3.1 Tasks und Futures

Die Klasse `Task` bietet eine komfortable Schnittstelle für die taskparallele Programmierung. Ihre Benutzung ist an die der `Thread`-Klasse angelehnt. So startet die Methode `Start` einen Task und `Wait`[2] wartet auf deren Beendigung:

```
Task t1 = new Task(DoSomething);
t1.Start();
// ...
t1.Wait();
```

Alternativ können Tasks über statische Methoden der Klasse `TaskFactory` erzeugt werden. Damit können wir unser Fibonacci-Beispiel aus Abschnitt 4.1.2 auf einfache Weise parallelisieren. Listing 12.13 zeigt, wie das geht.

```
int FibonacciTask(int n) {
  if (n < 2) {
    return n;
  }
  int x, y;
  var t1 = Task.Factory.StartNew(
    () => {x = FibonacciTask(n-1);}
  );
  var t2 = Task.Factory.StartNew(
    () => {y = FibonacciTask(n-2);}
  );
  t1.Wait();
  t2.Wait();
  return x + y;
}
```

Listing 12.13 *Parallele Berechnung der Fibonacci-Zahlen mit Tasks*

Mit `Task.WaitAll` kann man auf mehrere Tasks warten. So könnten wir anstatt der beiden `Wait`-Aufrufe in Listing 12.13 auch `Task.WaitAll(t1, t2)` schreiben. Tasks können ebenso wie Threads nicht mehrfach gestartet oder wiederverwendet werden.

Die statische Methode `Parallel.Invoke` erlaubt es, ein »Bündel« von Funktionen parallel auszuführen. Sie kehrt erst zurück, wenn die Funktionen beendet sind. Ein Aufruf von `Wait` oder `WaitAll` ist daher nicht notwendig. Listing 12.14 zeigt das entsprechend angepasste Fibonacci-Beispiel.

Die generische Variante `Task<TResult>` repräsentiert einen Task mit einem Rückgabewert. Das Komfortable dabei ist, dass der Zugriff auf die `Result`-

[2] Sofern es der verwendete Scheduler unterstützt, führt `Wait` noch nicht bearbeitete Tasks im aktuellen Thread aus. Das ist das Standardverhalten. Es ist jedoch möglich, eigene Scheduler zu implementieren, die sich anders verhalten.

```
int FibonacciInvoke(int n) {
  if (n < 2) {
    return n;
  }
  int x, y;
  Parallel.Invoke(
    () => {x = FibonacciInvoke(n-1);},
    () => {y = FibonacciInvoke(n-2);}
  );
  return x + y;
}
```

Listing 12.14 *Parallele Berechnung der Fibonacci-Zahlen mit* `Parallel.Invoke`

Property des Tasks so lange blockiert, bis das Ergebnis vorliegt. Auf diese Weise unterstützen .NET-Tasks das Future-Konzept (siehe Abschnitt 6.1.3). Betrachten wir noch einmal die Berechnung der Fibonacci-Zahlen. Wir müssen in jeder Rekursion warten, bis die Ergebnisse vorliegen. Das können wir, wie in Listing 12.15 gezeigt, mit Futures erreichen.

```
public static int FibonacciFuture(int n) {
  if (n < 2) {
    return n;
  }
  var t1 = Task<int>.Factory.StartNew(
    () => FibonacciFuture(n-1)
  );
  var t2 = Task<int>.Factory.StartNew(
    () => FibonacciFuture(n-2)
  );
  return t1.Result + t2.Result;
}
```

Listing 12.15 *Parallele Berechnung der Fibonacci-Zahlen mit Futures*

Doch nicht immer ist es notwendig, unmittelbar nach dem Start paralleler Aktivitäten auf die Ergebnisse zu warten. Erinnern wir uns an den Quicksort-Algorithmus aus Abschnitt 4.1.2. Bei diesem Algorithmus genügt es zu wissen, wann das gesamte Feld sortiert ist, also alle Task beendet sind. Das lässt sich mithilfe von Eltern-Kind-Beziehungen erreichen. In .NET werden Kind-Tasks aus dem Kontext eines laufenden Tasks mit der Option `AttachedToParent` erzeugt. Listing 12.16 zeigt die Implementierung des Quicksort-Algorithmus.

Neben `AttachedToParent` gibt es weitere Optionen, die man beim Erzeugen eines Tasks angeben kann (alle Optionen sind Werte des Aufzählungstyps `TaskCreationOptions`). Wir hatten schon erwähnt, dass der Task Scheduler von .NET mithilfe einer Heuristik die Anzahl der Threads an die Auslastung des Sys-

```
public static void QuicksortParallel(int[] a) {
  // Wurzeltask:
  Task t = Task.Factory.StartNew(
    () => QuicksortTask(a, 0, a.Length)
  );
  t.Wait();
}

private static void QuicksortTask(int[] a,
                                  int from, int to) {
  if (to - from > GRAINSIZE) {
    int pivot = Partition(a, from, to);
    Task.Factory.StartNew(
      () => QuicksortTask(a, from, pivot),
      TaskCreationOptions.AttachedToParent
    );
    Task.Factory.StartNew(
      () => QuicksortTask(a, pivot + 1, to),
      TaskCreationOptions.AttachedToParent
    );
  } else {
    // sequenziell sortieren, z.B. mit InsertionSort
    InsertionSort(a, from, to);
  }
}
```

Listing 12.16 *Parallele Implementierung von Quicksort mit Eltern-Kind-Tasks*

tems anpasst. Wenn man im Voraus weiß, dass bestimmte Tasks sehr lange laufen werden, kann man dem Scheduler über die Option `LongRunning` einen entsprechenden Hinweis darauf geben. Diese Information fließt in die Heuristik ein. Der Scheduler kann für solche Tasks dann weitere Threads erzeugen.

Die Option `PreferFairness` beeinflusst die Scheduling-Strategie des Task-Schedulers. Ist die Option aktiviert, werden früh gestartete Tasks mit hoher Wahrscheinlichkeit vor später gestarteten Tasks abgearbeitet. Es wird allerdings keine Reihenfolge garantiert. Ist diese Option deaktiviert, können einzelne Tasks aufgrund der Arbeitsweise des Work-Stealing-Verfahrens durch später erzeugte Tasks verdrängt werden.

12.3.2 Fortsetzungstasks

In vielen Fällen möchte man über das Ende einer asynchronen Operation informiert werden. Dazu kann man zum Beispiel eine Rückrufmethode an einen Task übergeben, die am Ende des Tasks aufgerufen wird (siehe Abschnitt 12.1.2). Eine andere Möglichkeit ist, für die Operationen, die nach einem Task ausgeführt

```
var task = Task<int>.Factory.StartNew(
  // ... Berechnung
  () => 42
);
var continuation = task.ContinueWith(
  (antecedent) => {
    Console.WriteLine("Die Antwort ist {0}", antecedent.Result);
  },
  TaskContinuationOptions.OnlyOnRanToCompletion
);
continuation.Wait();
```

Listing 12.17 *Fortsetzungstask mit Bedingung*

werden sollen, wiederum einen Task zu erstellen. Das lässt sich mit sogenannten Fortsetzungstasks bewerkstelligen, die nach dem Ende eines Tasks in die Warteschlangen des Schedulers eingereiht werden. Dabei kann man eine Bedingung angeben, anhand derer entschieden wird, ob der Fortsetzungstask ausgeführt wird. Die Bedingung wird in Form von `TaskContinuationOptions` angegeben. Die am häufigsten verwendeten Bedingungen sind:

`OnlyOnCanceled/NotOnCanceled` Der Fortsetzungstask wird (nicht) ausgeführt, wenn der Vorgängertask abgebrochen wurde (wie man Tasks abbricht, behandeln wir in Abschnitt 12.6).

`OnlyOnFaulted/NotOnFaulted` Der Fortsetzungstask wird (nicht) ausgeführt, wenn der Vorgängertask eine nicht behandelte Ausnahme geworfen hat.

`OnlyOnRanToCompletion/NotOnRanToCompletion` Wenn der Vorgängertask bis zum Ende ausgeführt wurde, d. h., er weder abgebrochen noch durch eine Ausnahme beendet wurde, wird der Fortsetzungstask (nicht) ausgeführt.

Fortsetzungstasks werden der Methode `ContinueWith` der Klasse `Task` übergeben. Bei dem Beispiel in Listing 12.17 wird der Fortsetzungstask nur dann ausgeführt, wenn der Vorgängertask seine Arbeit erfolgreich verrichtet hat. Der Vorgängertask wird als Argument übergeben (`antecedent`), um Daten an den Fortsetzungstask weiterzureichen. Ein Fortsetzungstask kann mehrere Vorgänger haben. Diese gibt man mit der statischen Methode `ContinueWhenAll` an.