Kapitel 3

Datenparallelität

In diesem Kapitel:

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

Nachdem Sie dieses Kapitel durchgearbeitet haben, werden Sie in der Lage sein,

- Daten- und Aufgabenparallelität zu unterscheiden,
- eine sequenzielle Schleife zu parallelisieren,
- parallele Schleifen ordnungsgemäß abzubrechen,
- unbehandelte Ausnahmen in parallelen Schleifen abzufangen,
- Reduktionen bei gleichzeitiger Minimierung der Abhängigkeiten durchzuführen und
- anderen das MapReduce-Entwurfsmuster zu erläutern

Im vorigen Kapitel haben wir uns mit dem Thema der Aufgabenparallelität befasst, also mit dem Ausführen separater, paralleler Aufgaben. Im Gegenzug dazu geht es bei der Datenparallelität um die Anwendung einer allgemeinen Operation auf die einzelnen Elemente einer Datensammlung – wie z.B. ein Preisnachlass von 10 Prozent auf alle Waren, die seit mehr als 90 Tage im Lager stehen. In diesem Fall wäre der Preisnachlass die allgemeine Operation und das Lager die Datensammlung. Es liegt in der Natur der Sache, dass Datenparallelität vor allem in datenorientierten und rechenintensiven Szenarien von Nutzen ist, also z.B. in Bereichen wie Rechnungswesen, Datenbankverwaltung, Wettervorhersagen, Verkaufsanalysen, wissenschaftlichen Auswertungen oder auch einer virtuellen Sportliga. Auf der anderen Seite gibt es in den meisten Anwendungen datenfokussierte Teile, und somit auch Gelegenheiten für die Implementierung von Datenparallelität.

Chancen für die sinnvolle Nutzung von Datenparallelität zu entdecken, fällt in der Regel leichter als die Suche nach Aufgabenparallelität: Halten Sie einfach nach Schleifen Ausschau. Schleifen sind anhand ihrer Syntax leicht aufzuspüren – Sie brauchen nur nach den einleitenden Schlüsselwörtern `for`, `foreach`, `while` oder `do while` zu suchen. Die `for`- und `foreach`-Anweisungen werden typischerweise für einfach aufgebaute, geradlinige Schleifen verwendet, während `while`-Schleifen eher in komplexeren Szenarien mit Abhängigkeiten zum Einsatz kommen. (Für Letztere gibt es in der parallelen Programmierung spezielle Entwurfsmuster.) Daneben gibt es natürlich auch noch andere Gelegenheiten für Datenparallelität – vielleicht wird in Ihrem Code ja eine Methode aufgerufen, die über eine Baum-Datenstruktur iteriert –, aber das Gros der möglichen Gelegenheiten werden Sie vermutlich in den Schleifen finden. Wenn die Iterationen einer Schleife voneinander unabhängig oder nur minimal abhängig sind, sollte es möglich sein, die Schleifeniterationen in parallele Operationen umzuwandeln. Wenn die Iterationen komplett unabhängig voneinander sind, sind sie *embarassingly parallel* und können mit optimaler Leistungssteigerung parallel ausgeführt werden.

Viele Schleifen iterieren über Auflistungen und führen dabei auf jedem Element die gleiche Operation aus. Datenparallelität ist überhaupt häufig mit der Parallelisierung von Schleifen verbunden, weswegen man auch manchmal von *Schleifenparallelität* spricht. Greifen wir noch einmal das obige Beispiel mit dem 10 %-igen Preisnachlass auf bestimmte Waren auf. Die Reduzierung des Preises ist eine unabhängige Operation, d.h., die Reduzierung einer Ware hat keine Auswirkung auf die anderen Waren, ist also *embarassingly parallel*. Folglich kann die Operation einfach dadurch parallelisiert werden, dass jede einzelne Preisreduzierung als separate Aufgabe ausgeführt wird. Die tatsächliche Granularität der Parallelisierung wird in der TPL (Task Parallel Library) durch den Standardpartitionierer festgelegt, der für die Partitionierung der Aufgaben verantwortlich ist. Die resultierenden Aufgaben werden dann über Threads aus dem Threadpool von Microsoft .NET Framework 4 ausgeführt. Wenn Sie auf diese Vorgänge Einfluss nehmen möchten, müssen Sie eigene Taskplaner und Partitionierer implementieren (siehe Kapitel 6).

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

Sequenzielle Schleifen in parallele Aufgaben umwandeln

Datenparallelität beginnt typischerweise mit dem »Aufdröseln« einer Schleife. Nehmen Sie z.B. die nachfolgend definierte, sequenzielle Schleife, deren Schleifenkörper 100 Mal durchlaufen wird. Die Ausführungszeit für diese `for`-Schleife ist die Summe der Ausführungszeiten der 100 Operationen. Geht man davon aus, dass die einzelne Operation 100 Millisekunden benötigt, ergibt sich also eine Gesamtausführungszeit von 10 Sekunden (100 × 100 Millisekunden).

```
for (int count = 0; count < 100; ++count)
{
   Operation();
}
```

Bedauerlich ist, dass der obige Code die Rechenleistung der verfügbaren Prozessoren nicht voll ausnutzt, siehe Abbildung 3.1:

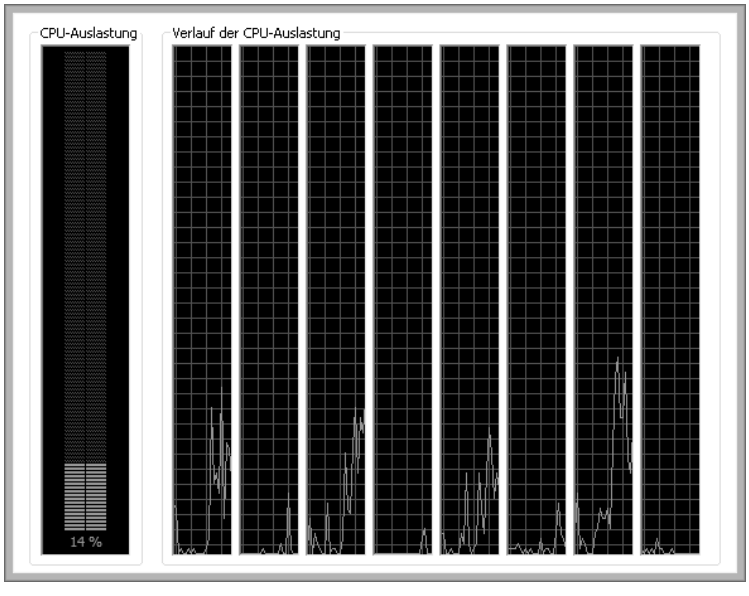

Abbildung 3.1 Ausführung der `for`-Schleife auf einem Computer mit acht Prozessorkernen

HINWEIS Um den Windows Task-Manager aufzurufen, drücken Sie die Tastenkombination Strg + ⇧ + Esc. Danach können Sie mit einem Doppelklick in irgendeines der angezeigten CPU-Diagramme in die spezielle CPU-Ansicht wechseln.

Wie Sie der Abbildung entnehmen können, ist die durchschnittliche Auslastung der Kerne relativ niedrig. Manche Kerne sind praktisch beschäftigungslos. Mit einem Wort: *Hier bleibt viel Prozessorleistung ungenutzt!*

Für alle diejenigen Leser, die das Beispiel selbst ausprobieren möchten, hier der vollständige Quellcode:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
```

```
namespace ParalleleSchleife
{
    class Program
    {
        static void Operation()
        {
            Thread.SpinWait(int.MaxValue);
        }

        static void Main(string[] args)
        {
            for (int count = 0; count < 100; ++count)
            {
                Operation();
            }
        }
    }
}
```

Erfreulicherweise bedarf es in den meisten Fällen nur weniger Änderungen, um eine sequenzielle Schleife in eine parallele Schleife zu verwandeln. Die parallele Version der obigen Schleife sieht beispielsweise wie folgt aus:

```
Parallel.For(1, 100, (count) =>
{
   Operation();
});
```

Das war nicht schwer! Und sehen Sie sich den Balken zur CPU-Auslastung an (Abbildung 3.2). Der Balken zeigt eine 100 %-ige Prozessorauslastung für unsere parallele Schleife.

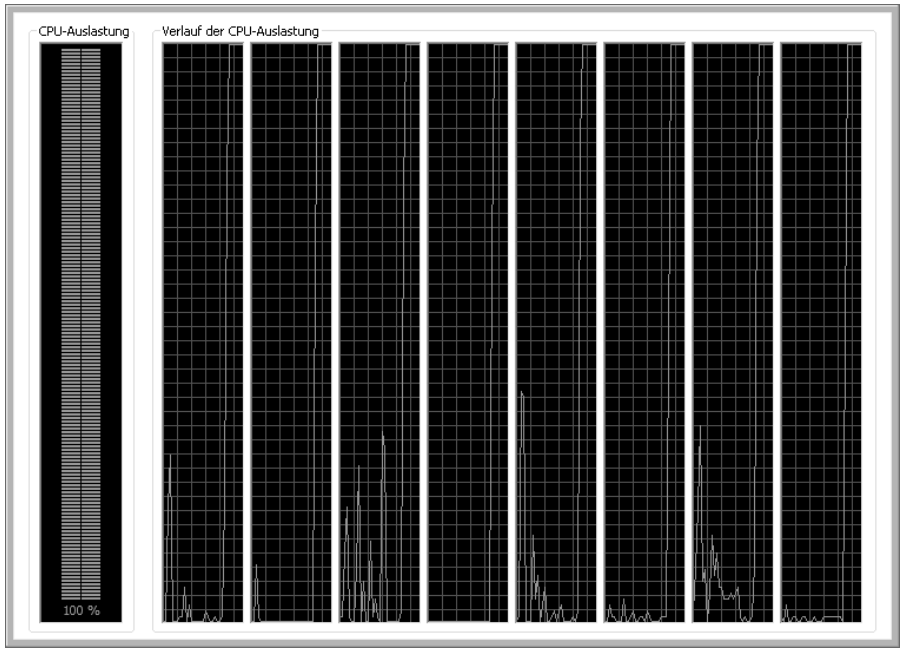

Abbildung 3.2 Nach dem Starten der Anwendung werden 100 Prozent CPU-Auslastung erreicht (Die kaum zu sehende waagerechte Linien am oberen Rand der einzelnen Graphen stehen für 100 %-ige Auslastung der zugehörigen Prozessorkerne.)

Lassen Sie uns die Besprechung der syntaktischen Feinheiten von `Parallel.For` noch etwas hintenanstellen. Viel interessanter ist im Moment, dass wir es mit dem obigen Code tatsächlich geschafft haben, die Operation in eine Handvoll von Aufgaben zu parallelisieren, die auf die verfügbaren Prozessorkerne verteilt ausgeführt werden. Wie viele Aufgaben es genau sind, wissen wir nicht, aber es sind auf jeden Fall genügend, um die Prozessorkerne beschäftigt zu halten – ein wichtiges Ziel der parallelen Programmierung.

Wie effizient sich Datenparallelität nutzen lässt, hängt entscheidend von der Unabhängigkeit der Schleifeniterationen ab. Unabhängige Iterationen führen zu parallelen Aufgaben ohne Abhängigkeiten, parallele Aufgaben ohne Abhängigkeiten führen zu hocheffizienter Datenparallelität. Manche Abhängigkeiten lassen sich mit minimaler Beeinträchtigung der Leistungssteigerung umsetzen – so z.B. die Reduktion, bei der eine Reihe von Operationen auf einen skalaren Wert reduziert wird (mehr hierzu weiter unten in diesem Kapitel). Manchmal findet man Abhängigkeiten, wo man sie gar nicht vermutet, wie z.B. in:

- Indizes
- gemeinsam genutztem Arbeitsspeicher
- gemeinsam genutzten Dateien
- vorgegebenen Abfolgen

Erkennen Sie die Abhängigkeiten in der folgenden Schleife? Der Code implementiert eine parallele Schleife, die den aktuellen Index quadriert und das Ergebnis in eine Datei schreibt. Das Ergebnis ist allerdings unerwartet – und das Resultat einer gar nicht so versteckten Abhängigkeit.

```
StreamWriter sw = new StreamWriter(filename);
Parallel.For (0, 100, (i)=>sw.WriteLine(i * i));
sw.Close();
```

Der Grund für das ganze Ungemach ist natürlich der Verweis auf die gemeinsam genutzte Datei. Wir können nicht ausschließen, dass der Code gleichzeitig aus zwei oder mehr parallel ausgeführten Aufgaben auf die Datei zuzugreifen versucht – und Dateien sind nicht implizit threadsicher. Hinzu kommt, dass die im Code verwendete Methode `Parallel.For` keine Zusagen darüber macht, in welcher Reihenfolge die gestarteten Aufgaben ausgeführt werden. Das heißt, es gibt keine Garantie, dass die Quadratwerte in aufsteigender Reihenfolge in die Datei geschrieben werden (wie es bei einer sequenziellen Schleife der Fall wäre). Hier ein Auszug aus der Datei, deren größter Wert 10.000 (100×100) lauten sollte:

```
6084
4900
705662415041
7225
6400
9216
73966561
```

Keiner der abgedruckten Werte ist kleiner als 1.000! Ein Umstand, der auf Datenverfälschungen durch gleichzeitige Schreibzugriffe zurückzuführen ist und belegt, welches Risiko von nicht behandelten Abhängigkeiten ausgeht. Zu unserem Leidwesen sind Abhängigkeiten nicht immer so leicht zu erkennen wie die gemeinsam genutzte Datei aus dem Beispiel. Die meisten Abhängigkeiten sind weit weniger offensichtlich und oft bleiben sie sogar unentdeckt. Umso wichtiger ist es, den parallelisierten Code rigoros zu testen: mit Unit-Tests, Stress-Tests, Concurrency-Tests. Das Umschreiben des Codes mag einfach sein, die Wahrung der Korrektheit ist es nicht.

Es gibt diverse Techniken und Tricks, wie man Abhängigkeiten erkennen kann. Eine dieser Techniken besteht darin, die Abfolge der Iterationen in einer Schleife umzudrehen, also z.B. von aufsteigend in absteigend. Liefert das Programm danach andere Ergebnisse oder stürzt es gar ab, ist dies ein starker Hinweis auf eine unentdeckte Abhängigkeit.

Wann lohnt sich die Parallelisierung?

Nicht jede sequenzielle Schleife sollte in parallele Aufgaben aufgedröselt werden. Ein Kriterium für oder wider die Parallelisierung ist die zu erwartende Leistungssteigerung.

Wenn die vorgesehenen Aufgaben relativ kurz sind, kann der mit der parallelen Ausführung verbundene Overhead – die Ausführung über den Threadpool, Kontextwechsel und andere Kosten – größer werden als die durch die Parallelisierung zu erhoffende Leistungssteigerung. Führen Sie daher unbedingt Benchmark-Tests durch, um sichergehen zu können, dass die anvisierten Leistungsverbesserungen auch tatsächlich erreicht werden. Stellt sich dabei heraus, dass der Leistungszuwachs nur minimal ausfällt, verzichten Sie auf die Parallelisierung oder versuchen Sie es mit einer anderen Blockgröße. Die Standardblockgröße wird von dem Standardpartitionierer der TPL vorgegeben. Größere Blöcke führen dazu, dass die einzelnen Aufgaben mehr Arbeit leisten müssen. Dies wiederum senkt die Kosten für die Parallelisierung und führt – hoffentlich – zu einer Verbesserung der Gesamtleistung.

Bei der Datenparallelisierung wird typischerweise ein und dieselbe Operation auf verschiedenen Daten ausgeführt. Identische Operationen bedeuten aber nicht zwangsweise auch identische Laufzeiten. Betrachten Sie dazu den folgenden Code, der eine Folge von Primzahlen ausgibt.

```
Parallel.For(1, 1000, (index) =>
{
   if(IsPrime(index))
   {
      Console.WriteLine(index);
   }
});
```

Alle Schleifeniterationen führen dieselbe Operation durch und trotzdem kann es drastische Unterschiede in den Laufzeiten der einzelnen Aufgaben geben, wenn die Implementierung der Methode `IsPrime()` für die Überprüfung großer Zahlen wie z.B. 1000 wesentlich länger benötigt als für die Überprüfung kleinerer Zahlen wie z.B. 81. Derartige Unwuchten in der Lastverteilung können zu ineffizienter Parallelisierung führen. Die Lösung ist dann in der Regel die Implementierung eines eigenen Partitionierers, der die Blockgrößen mithilfe eines gewichteten Algorithmus so variiert, dass die Last möglichst gleichmäßig auf die Prozessoren verteilt wird.

HINWEIS Denken Sie daran, dass die Methode `Parallel.For` die Primzahlen nicht notwendigerweise in aufsteigender Reihenfolge berechnet. Die zur Ausgabe verwendete Methode `Console.WriteLine` wird übrigens intern synchronisiert, sodass die Ausgabe threadsicher ist.

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

Die parallele for-Schleife

Die for-Schleife ist wohl programmiersprachenübergreifend die am häufigsten genutzte Anweisung für Iterationen. Das folgende Beispiel zeigt eine serielle for-Schleife, welche die einzelnen Iterationen in der vorgegebenen Reihenfolge durchführt. Die Schleife iteriert von 0 bis 1000 und führt bei jedem Schleifendurchgang die gewünschte Operation durch. Wenn der Wert der Schleifenvariable count größer oder gleich 1000 ist, wird die Schleife beendet.

```
for (int count = 0; count < 1000; ++count)
{
   DoSomething();
}
```

Das TPL-Pendant zur for-Schleife ist die Methode Parallel.For , welche die Iterationen nicht mehr sequenziell, sondern parallel ausführt. Die Klasse Parallel, die Sie im Namespace System.Threading.Tasks finden, definiert mehrere Überladungen der Parallel.For-Methode. Die ersten beiden Parameter der Basisüberladung sind der Startwert und der obere (exklusive) Grenzwert. Das Inkrement ist standardmäßig 1. Der letzte Parameter ist ein Action-Delegat. Für diesen Parameter können Sie einen Delegaten, einen Lambda-Ausdruck oder auch eine anonyme Methode übergeben, die den aktuellen Index als einziges Argument übernimmt. Parallel.For liefert als Ergebnis eine ParallelLoopResult-Struktur zurück, in der der Status der Parallel.For-Schleife abgespeichert ist. Hier der Prototoyp der Parallel.For-Methode:

```
public static ParallelLoopResult For(
   int fromInclusive,
   int toExclusive,
   Action<int> body
)
```

Das nächste Beispiel zeigt eine Parallel.For-Schleife, die eine Operation 100-mal ausführt. Anders als die Iterationen der for-Schleife, werden die parallelen Iterationen nicht notwendigerweise in aufsteigender Reihenfolge ausgeführt, sodass die siebenhundertste Iteration möglicherweise vor der zehnten Iteration ausgeführt wird. Sie werden aber auf jeden Fall alle ausgeführt – sofern Sie nicht die Schleife mittels einer ParallelLoopState.Break- oder ParallelLoopState.Stop-Anweisung vorzeitig abbrechen.

```
Parallel.For(0, 100, (count) =>
{
   DoSomething();
});
```

Die Methode Parallel.ForEach ist das parallele TPL-Äquivalent zur foreach-Anweisung von Microsoft Visual C#. Verwenden Sie die Methode Parallel.ForEach, wenn Sie eine Operation parallel auf den Elementen einer Auflistung ausführen möchten. Der erste Parameter der Basisüberladung ist die Auflistung. Der nächste Parameter ist ein Action-Delegat und gibt die Operation an, die auf die Elemente in der Auflistung angewendet werden soll. Der Action-Delegat übernimmt als einziges Argument das aktuelle Element.

```
public static ParallelLoopResult ForEach<TSource>(
   IEnumerable<TSource> source,
   Action<TSource> body
)
```

Die normale foreach-Schleife sieht wie folgt aus und wird natürlich sequenziell ausgeführt:

```
foreach (int item in aList)
{
  Operation(item);
}
```

Und hier die gleiche Schleife unter Verwendung der Methode Parallel.ForEach. In diesem Fall ist jede Iteration eine parallele Aufgabe, die nicht sequenziell, sondern parallel ausgeführt wird.

```
Parallel.ForEach(aList, (item)=> {
   Operation(item);
} );
```

Um etwas mehr Praxis im Umgang mit parallelen Schleifen zu gewinnen, stellen Sie sich einmal vor, Sie wären stolzer Besitzer eines Ladens mit angeschlossenem Warenlager. Einmal im Monat gehen Sie das Warenlager durch und korrigieren die Preise der Waren, die schon mehr als 90 Tage im Lager sind. Waren, die weniger als 500 € kosten, reduzieren Sie um 10 %, höherpreisige Waren um 20 %. Höherpreisige Waren haben eine höhere Gewinnspanne.

Erzeugen Sie eine Parallel.For-Schleife zur Reduzierung der Warenpreise

1. Legen Sie mit Visual Studio 2010 eine neue C#-Konsolenanwendung an. Fügen Sie mithilfe der using-Anweisung der Liste der Namespaces noch den Namespace System.Threading.Tasks hinzu. Definieren Sie innerhalb der Klasse (über der Main-Methode) ein statisches int-Array mit den aktuellen Preisen der Waren, die schon länger als 90 Tage auf Lager liegen.

   ```
   static int[] inventoryList = new int []
      {100, 750, 400, 75, 900, 975, 275, 750, 600, 125, 300};
   ```

2. Definieren Sie in der Main-Methode eine Parallel.For-Schleife zum Durchlaufen der Inventarliste.

   ```
   Parallel.For( 0, inventoryList.Length, (index) => {
   ```

3. Nun zur parallelen Operation. Definieren Sie eine temporäre Variable, in der Sie den Preis der aktuellen Ware speichern. Wenn die Ware mehr als 500 € kostet, reduzieren Sie den Preis um 20 %, andernfalls um 10 %.

   ```
   var price= inventoryList[index];
   if (price> 500)
   {
      inventoryList[index] = (int)(price* .8);
   }
   else
   {
      inventoryList[index] = (int)(price* .9);
   }
   ```

4. Geben Sie mithilfe von Console.WriteLine den korrigierten Preis aus.
5. Fügen Sie am Ende der Main-Methode einen Console.ReadLine-Aufruf ein, damit man die Ausgaben in Ruhe lesen kann, bevor das Konsolenfenster geschlossen wird. Geben Sie vielleicht vorab noch einen erklärenden Hinweis aus.

   ```
   Console.WriteLine("Zum Beenden die Eingabetaste drücken");
   Console.ReadLine();
   ```

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

6. Erstellen Sie die Anwendung und führen Sie sie aus.

Hier noch einmal der vollständige Quellcode der Anwendung:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Preisnachlass
{
   class Program
   {
      static int[] inventoryList = new int[] {100, 750, 400, 75, 900, 975, 275,
            750, 600, 125, 300};

      static void Main(string[] args)
      {
         Parallel.For(0, inventoryList.Length, (index) =>
         {
            var price = inventoryList[index];
            if (price > 500)
            {
               inventoryList[index] = (int)(price * .8);
            }
            else
            {
               inventoryList[index] = (int)(price * .9);
            }

            Console.WriteLine("Ware {0,4} Preis: {1, 7:f}",
                              index, inventoryList[index]);
         });
      }
   }
}
```

Listing 3.1 *Program.cs* aus dem Projekt *Preisnachlass.csproj*

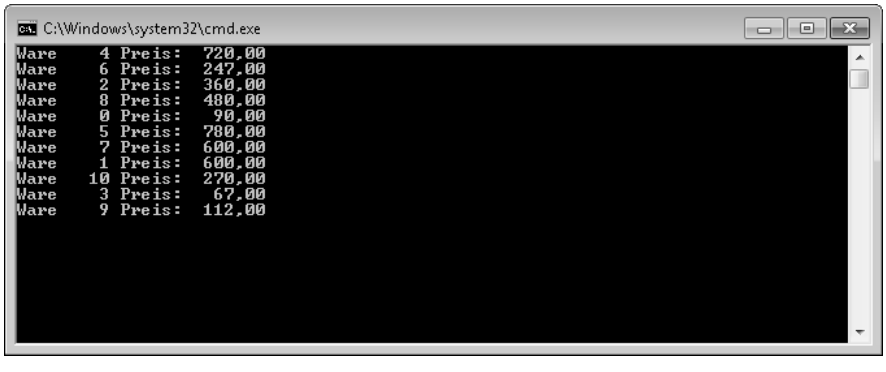

Abbildung 3.3 Die Ausgabe der Anwendung. Wie Sie sehen, werden die Waren nicht in der durch das Array vorgegebenen Reihenfolge aufgeführt. (Beachten Sie auch, dass Ihre Ausgabe von der abgebildeten Ausgabe abweichen kann, da die Reihenfolge, in der die einzelnen Aufgaben der `Parallel.For`-Schleife ausgeführt werden, von Programminstanz zu Programminstanz variieren kann.)

ACHTUNG Wenn Sie Konsolenanwendungen im Debug-Modus ausführen oder per Doppelklick im Windows Explorer starten, wird das Konsolenfenster nach Beendigung der Anwendung sofort geschlossen und es bleibt möglicherweise keine Zeit, die Ausgabe zu lesen. Um zu verhindern, dass das Konsolenfenster automatisch geschlossen wird, können Sie am Ende des Codes einen `Console.ReadLine`-Aufruf einfügen – optional kombiniert mit einer `Console.WriteLine`-Ausgabe, die den Benutzer darauf hinweist, dass er zum Beenden die ↵-Taste drücken soll. Alternativ können Sie die Anwendung aber auch im Release-Modus ausführen, indem Sie Strg + F5 drücken. Das Einfügen einer eigenen Anweisung zum Offenhalten des Konsolenfensters ist dann nicht mehr erforderlich.

Schleifen abbrechen

Die normalen `for`- und `foreach`-Schleifen von C# kennen zwei spezielle Abbruchbefehle: `break` und `continue`. Mit der `break`-Anweisung brechen Sie die aktuelle Iteration ab und verlassen danach die Schleife. Mit der `continue`-Anweisung überspringen Sie den Rest der aktuellen Iteration und fahren danach mit der nächsten Iteration fort. Das Abbrechen einer `Parallel.For`- oder `Parallel.ForEach`-Schleife ist etwas komplizierter, da diese ja nicht sequenziell ausgeführt werden. Dies beginnt damit, dass die `break`- und `continue`-Anweisungen in parallelen `for`-Schleifen nicht verwendet werden, da es sich bei `Parallel.For` und `Parallel.ForEach` ja um Methoden und nicht um in der Sprache verankerte Schleifenkonstrukte handelt. Stattdessen gibt es für den Abbruch paralleler Schleifen spezielle Konstrukte.

Um eine parallele Schleife abbrechen zu können, müssen Sie dem `Action`-Delegaten für die parallele Operation als zweites Argument ein `ParallelLoopState`-Objekt übergeben. Danach können Sie die Schleife mithilfe der Methode `ParallelLoopState.Break` abbrechen. Zum Zeitpunkt des Abbruchs können andere Aufgaben schon beendet, noch am Laufen oder noch gar nicht gestartet worden sein. Bei länger laufenden Aufgaben sollten Sie in regelmäßigen Abständen prüfen, ob ein Abbruch gewünscht wird. Fragen Sie dazu einfach die Eigenschaft `ParallelLoopState.ShouldExitCurrentIteration` ab. Ist der Wert `true`, wurde der Abbruch der Schleife gefordert. In diesem Fall können Sie von der Eigenschaft `ParallelLoopState.LowestBreakIteration` den Index der abbrechenden Aufgabe abfragen. Aufgaben mit einem höheren Index sollten sich danach bei Gelegenheit freiwillig beenden. Aufgaben mit einem niedrigeren Index können ganz normal weiterlaufen. Aufgaben mit einem niedrigeren Index, die noch nicht gestartet wurden, dürfen ganz normal starten und ausgeführt werden. Aufgaben mit einem höheren Index, die noch nicht gestartet wurden, werden nicht mehr ausgeführt.

Abbildung 3.4 illustriert die Wirkungsweise der Methode `ParallelLoopState.Break` anhand eines fiktiven Szenarios. Die Ergebnisse können abhängig von verschiedenen Faktoren variieren. In Phase 1 trägt die Methode `Parallel.For` sechs Aufgaben zur Ausführung in die Wartschlange des Threadpools von .NET Framework 4 ein. Die Aufgaben 1, 2, 4 und 5 werden gestartet. Für die Aufgaben 3 und 6 gibt es keine verfügbaren Prozessorkerne. Am Ende von Phase 2 ruft Aufgabe 4 die Methode `ParallelLoopState.Break` auf, um die Schleife abzubrechen. In Phase 3 schließlich werden die Aufgaben 1, 2 und 3 trotz des Schleifenabbruchs noch zu Ende ausgeführt, da ihre Indizes kleiner sind als der Index der abbrechenden Aufgabe. Aus diesem Grund wurde Aufgabe 3 auch noch der Start erlaubt. Aufgabe 5 dagegen hat entdeckt, dass die Schleife abgebrochen wurde und sich freiwillig beendet. Aufgabe 6, deren Index höher ist, darf nicht einmal mehr starten.

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

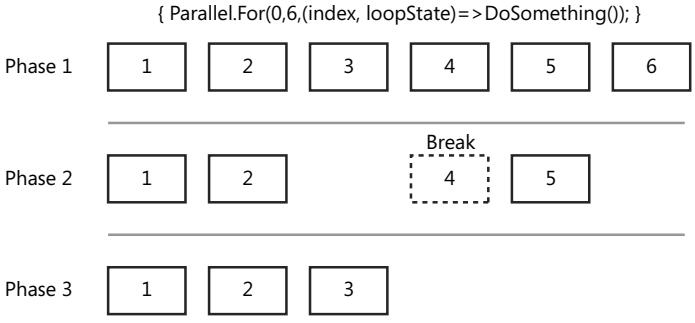

Abbildung 3.4 Abbruch einer parallelen Schleife mit `ParallelLoopState.Break`

Ein etwas anderes Abbruchmodell ist mit der Methode `ParallelLoopState.Stop` verknüpft. Beim Abbrechen einer Schleife mit `ParallelLoopState.Stop` wird von *allen* laufenden Aufgaben erwartet, dass sie sich bei passender Gelegenheit selbst beenden. Laufende Aufgaben können am Wert der Eigenschaft `ParallelLoopState.IsStopped` erkennen, ob das Beenden der Schleife veranlasst wurde. Der Wert der Eigenschaft ist dann `true` und die Aufgaben sollten sich wie gesagt bei Gelegenheit freiwillig selbst beenden. Noch nicht gestartete Aufgaben werden, anders als im Falle der Methode `ParallelLoopState.Break`, nicht mehr ausgeführt – gleichgültig, was für einen Index-Wert sie besitzen. Im Vergleich zu `ParallelLoopState.Break` werden also weniger Aufgaben gestartet oder normal zu Ende ausgeführt. `ParallelLoopState.Stop` ist somit das einfachere, klarere Abbruchmodell.

Abbildung 3.5 illustriert den Effekt der Methode `ParallelLoopState.Stop`. In Phase 1 werden sechs Aufgaben geplant, aber nicht gestartet. In Phase 2 werden die Aufgaben 1, 2, 4 und 5 ausgeführt. Dann ruft Aufgabe 4 die Methode `ParallelLoopState.Stop` auf. Die Aufgaben 1, 2 und 5 erkennen irgendwann, dass die Schleife abgebrochen wurde und beenden sich selbst. In Phase 3 gibt es keine laufenden Aufgaben mehr.

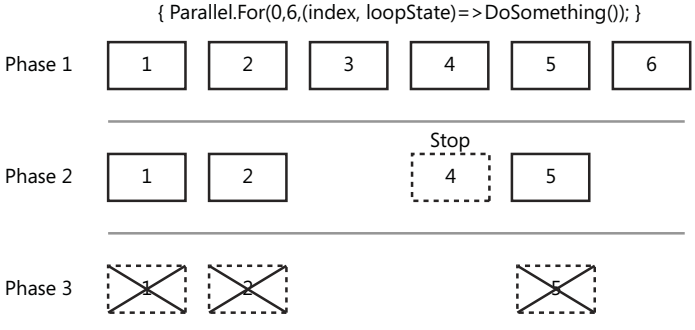

Abbildung 3.5 Abbruch einer parallelen Schleife mit `ParallelLoopState.Stop`

In der folgenden Übung werden Sie eine `Parallel.For`-Schleife starten und abbrechen. Der Index der Iteration, die den Abbruch veranlassen soll, wird als Befehlszeilenargument entgegengenommen.

Starten Sie eine Parallel.For-Schleife und brechen Sie sie vorzeitig ab

1. Legen Sie mit Visual Studio 2010 eine neue C#-Konsolenanwendung an. Fügen Sie `using`-Anweisungen für die Namespaces `System.Threading` und `System.Threading.Tasks` ein.
2. Definieren Sie über der `Main`-Methode eine statische Methode namens `HalfOperation`. Diese Methode soll die halbe Operation darstellen, die eine Iteration durchführt. Die Methode besitzt keine Parameter und liefert `void` zurück. Rufen Sie in der Methode `Thread.SpinWait` mit dem halben Maximalwert für `int` auf.

```
static void HalfOperation()
{
   Thread.SpinWait(int.MaxValue / 2);
}
```

3. In `Main` wandeln Sie das erste Befehlszeilenargument in einen Indexwert für den Schleifenabbruch um. Mithilfe der Methode `int.TryParse` lässt sich dies bequem bewerkstelligen, ohne dass ungültige Werte eine Ausnahme auslösen. Im Falle eines Scheiterns beenden Sie die Anwendung einfach mit `return`.

```
int cancelValue;
if(!int.TryParse(args[0], out cancelValue))
{
   return;
}
```

4. Beginnen Sie eine parallele Schleife mit einem Startwert von 0 und einem oberen Grenzwert von 20. Für die Schleifenoperation erzeugen Sie einen Lambda-Ausdruck, dem Sie den Schleifenindex und ein `ParallelLoopState`-Objekt übergeben.

```
Parallel.For(0, 20, (index, loopState) =>
```

5. In dem Lambda-Ausdruck geben Sie den Index der aktuellen Aufgabe aus und rufen die Methode `HalfOperation` auf. Als Nächstes prüfen Sie, ob dies die Aufgabe ist, die die Schleife abbrechen soll. Wenn ja, rufen Sie die Methode `ParallelLoopState.Break` auf. Danach geben Sie eine Meldung aus, dass die Schleife beendet wird, und stoppen die aktuelle Aufgabe.

```
Console.WriteLine("Aufgabe {0} wurde gestartet ...", index);
HalfOperation();
if (cancelValue == index)
{
   loopState.Break();
   Console.WriteLine("Schleife wird beendet. Aufgabe {0} abgebrochen...", index);
   return;
}
```

6. Aufgaben sollten regelmäßig überprüfen, ob ein Schleifenabbruch veranlasst wurde. Prüfen Sie zuerst durch Abfragen der Eigenschaft `ParallelLoopState.LowestBreakIteration.HasValue`, ob die Schleife abgebrochen werden soll. Wenn ja, fragen Sie den Index der abbrechenden Aufgabe ab. Ist dieser Index größer als der Index der aktuellen Aufgabe, beenden Sie die aktuelle Aufgabe. Und vergessen Sie nicht, entsprechende Meldungen auszugeben, damit Sie die Vorgänge in der Anwendung nachverfolgen können.

```
if (loopState.LowestBreakIteration.HasValue)
{
   if (index > loopState.LowestBreakIteration)
   {
      Console.WriteLine("Aufgabe {0} wurde abgebrochen", index);
      return;
   }
}
HalfOperation();
Console.WriteLine("Aufgabe {0} wurde beendet.", index);
```

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

Hier noch einmal der vollständige Quellcode der Anwendung:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace ParalleleSchleifeBreak
{
   class Program
   {
      static void HalfOperation()
      {
         Thread.SpinWait(int.MaxValue / 2);
      }
      static void Main(string[] args)
      {
         int cancelValue;

         if (!int.TryParse(args[0], out cancelValue))
         {
            return;
         }

         Parallel.For(0, 20, (index, loopState) =>
         {
            Console.WriteLine("Aufgabe {0} wurde gestartet ...", index);
            HalfOperation();
            if (cancelValue == index)
            {
               loopState.Break();
               Console.WriteLine("Schleife wird beendet. Aufgabe {0} abgebrochen...",
                                 index);
               return;
            }
            if (loopState.LowestBreakIteration.HasValue)
            {
               if (index > loopState.LowestBreakIteration)
               {
                  Console.WriteLine("Aufgabe {0} wurde abgebrochen", index);
                  return;
               }
            }
            HalfOperation();
            Console.WriteLine("Aufgabe {0} wurde beendet.", index);
         });
      }
   }
}
```

Listing 3.2 *Program.cs* aus dem Projekt *ParalleleSchleifeBreak.csproj*

In dem in Abbildung 3.6 dargestellten Programmlauf erfolgt der Abbruch der Schleife in Aufgabe 11. Zu diesem Zeitpunkt wurden bereits etliche weitere Aufgaben gestartet. (Die Ausgaben auf Ihrem Bildschirm dürften ähnlich aussehen.)

Abbildung 3.6 Die ersten Ausgaben der Anwendung

In Abbildung 3.7 sehen Sie die letzten Ausgaben der Anwendung. Beachten Sie, dass Aufgaben mit einem Index kleiner 11 auch nach dem Schleifenabbruch noch gestartet werden bzw. ganz normal zu Ende geführt werden.

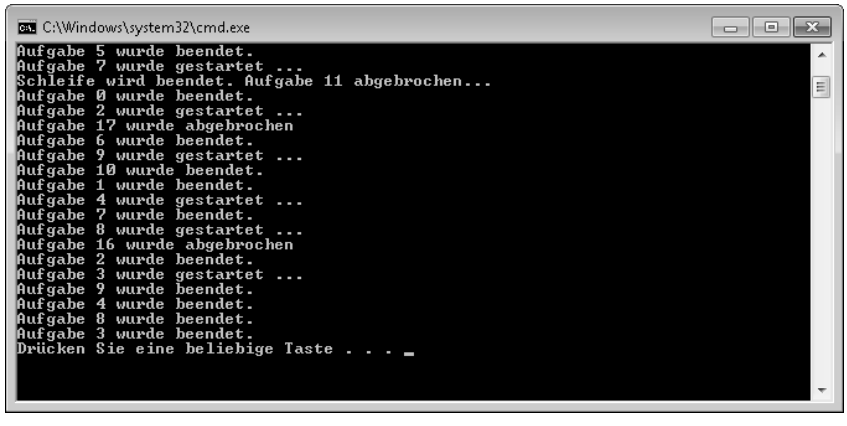

Abbildung 3.7 Die letzten Ausgaben der Anwendung

Ausnahmen abfangen

Sie können in parallelen Schleifen Ausnahmen auslösen, sollten dann aber folgende Punkte beachten:

- Parallele Aufgaben, die sich bereits in der Ausführung befinden, dürfen weiterlaufen, bis sie ihre Arbeit erledigt haben. Das heißt, Aufgaben können auch nach Auftreten einer unbehandelten Ausnahme noch weiter ausgeführt werden.
- Iterationen, die noch nicht gestartet wurden, werden nach Auftreten einer Ausnahme grundsätzlich nicht mehr gestartet
- Länger laufende Aufgaben sollten den Wert der Eigenschaft `ParallelLoopState.IsExceptional` überprüfen, um zu erkennen, wenn eine unbehandelte Ausnahme ausgelöst wurde. In diesem Fall lautet der Wert der Eigenschaft `true` und die Aufgabe sollte sich bei der nächsten Gelegenheit selbst beenden.
- Wegen der parallelen Ausführung der Aufgaben ist es möglich, dass mehr als eine Ausnahme ausgelöst wird. Aus diesem Grund löst die Methode eine `AggregateException`-Ausnahme aus. In deren `AggregateException.InnerExceptions`-Auflistung sind die einzelnen Ausnahmen gesammelt.

- Unbehandelte Ausnahmen aus einer parallelen Schleife sollten im Verbindungsthread behandelt werden. Steht der parallele Aufruf nicht in einem `try/catch`-Block, kann die Ausnahme zum Absturz der Anwendung führen.
- Unbehandelte Ausnahmen haben Vorrang vor `ParallelLoopState.Break` und `ParallelLoopState.Stop`

Das folgende Beispiel demonstriert, wie unbehandelte Ausnahmen aus parallelen Schleifen zu behandeln sind. Der Code ist so aufgebaut, dass er in der vierten Aufgabe absichtlich eine unbehandelte Ausnahme auslöst, die dann im `try/catch`-Block des Verbindungsthreads abgefangen wird. Wichtig ist, dass die `Parallel.For`-Anweisung innerhalb eines `try`-Blocks liegt, damit beim Auftreten einer unbehandelten Ausnahme die Programmausführung an den Verbindungsthread und schließlich an den `catch`-Filter für die `AggregateException`-Ausnahme weitergereicht wird. Innerhalb des `catch`-Blocks wird dann über die `AggregateException.InnerExceptions`-Auflistung iteriert und die unbehandelten Ausgaben werden ausgegeben.

```
try
{
   Parallel.For(0, 6, (index) =>
   {
      Console.WriteLine("Aufgabe {0} wurde gestartet.", index);
      if (index == 4)
      {
         throw new Exception();
      }
      DoSomething();
      Console.WriteLine("Aufgabe {0} wurde beendet.", index);
   });
}
catch (AggregateException ax)
{
   Console.WriteLine("\nFehlerliste: \n");
   foreach(var error in ax.InnerExceptions)
   {
      Console.WriteLine(error.Message);
   }
}
```

Listing 3.3 Auszug aus *Program.cs* (Projekt *ParallelForAusnahmen.csproj*)

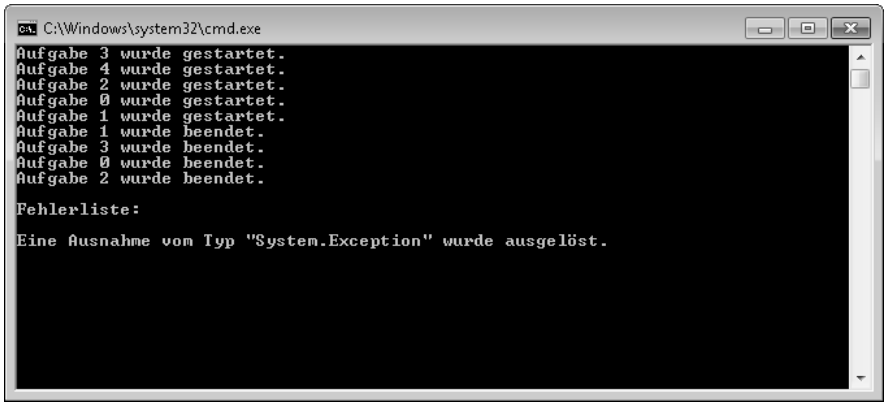

Abbildung 3.8 Ausgabe des Programms zur Ausnahmenbehandlung

Abhängigkeiten

Voneinander unabhängige Iterationen sind ideal für die Parallelisierung, da nur mit ihnen eine optimale Leistungssteigerung möglich ist. Doch nicht jede Schleife enthält perfekt unabhängige Iterationen. Dies gilt ganz besonders, wenn bestehende sequenzielle Schleifen, die von ihrem ursprünglichen Entwickler unter ganz anderen Vorgaben und Voraussetzungen erstellt wurden, nachträglich parallelisiert werden sollen. Abhängigkeiten, die nicht korrekt behandelt werden, können Ergebnisse verfälschen oder zum Absturz der Anwendung führen. Umgekehrt kann die Auflösung der Abhängigkeiten, die üblicherweise mit mehr oder weniger umfangreichen Synchronisierungen verbunden ist, die Leistung beeinträchtigen. Da für die meisten Anwendungen aber Korrektheit wichtiger ist als optimale Effizienz, sind die Alternativen in der Regel begrenzt.

Es gibt diverse Techniken, mit denen man Abhängigkeiten in parallelem Code auflösen kann. Eine detaillierte Besprechung dieser Techniken würde hier allerdings zu weit führen, weswegen wir uns auf die am weitesten verbreitete Technik, die Reduktion, konzentrieren werden.

Reduktion

Bei der Reduktion geht es darum, eine gegebene Menge von Werten auf einen Ergebniswert zu reduzieren. Beispielsweise, indem Sie die Summe der Werte berechnen. Werfen Sie einen Blick auf die folgende Schleife. Die Abhängigkeiten entstehen in diesem Fall aus der skalaren Variable `total`, die von allen Aufgaben genutzt wird. Mehr noch, die skalare Variable wird threadübergreifend gemeinsam genutzt, wobei jeder Thread eine oder mehrere parallele Aufgaben ausführt. Das Problem liegt also mehr in den Threads als in den Aufgaben.

```
int [] values=new int [] {1,2,3,4,5,6,7,8,9,10,
                          11,12,13,14,15,16,17,18,19,20};
int total = 0;
Parallel.ForEach(values, (item) =>
{
    total += item;
});
```

Die Herausforderung besteht nun darin, den obigen Code threadsicher zu machen, ohne auf der anderen Seite signifikante Leistungseinbußen in Kauf nehmen zu müssen.

In der TPL benutzen Sie für die Reduktion eine private threadlokale Variable. Da diese von den parallelen Aufgaben, die auf ein und demselben Thread ausgeführt werden, nacheinander (sequenziell) genutzt wird, ist ihre Verwendung in diesem Kontext threadsicher. Innerhalb der Aufgaben, die auf einem gemeinsamen Thread ausgeführt werden, bedarf es also keinerlei Synchronisierung.

Wenn die parallele Schleife endet, enthält jeder Thread ein Teilergebnis. Es wird dann eine spezielle Methode aufgerufen, deren Aufgabe es ist, die Teilergebnisse zu dem endgültigen Ergebnis zusammenzufassen. Dies ist die einzige Operation, die auf die globale Variable zugreift und synchronisiert werden muss.

ACHTUNG Die parallele Schleifenoperation muss sowohl kommutativ als auch assoziativ sein.

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

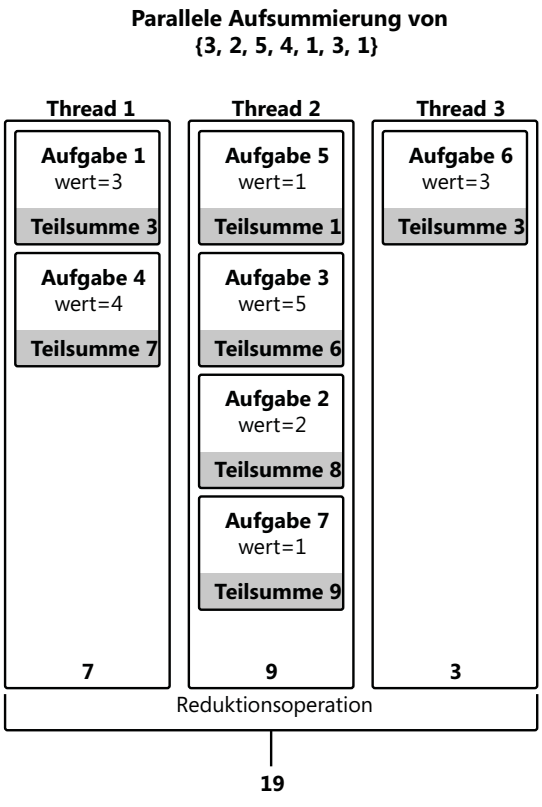

Abbildung 3.9 Reduktion für eine parallele Schleife

Wie bereits erwähnt, sollten Sie nur Operationen parallelisieren, die sowohl kommutativ als auch assoziativ sind. Ist die Operation *kommutativ*, bedeutet dies, dass die Operanden der Operation vertauscht werden können. (Vermutlich werden Sie sich von Ihrer Schulzeit her noch an das Kommutativgesetz erinnern, das meist als `a+b=b+a` formuliert wird.) Kommutative Operationen sind z.B. Addition und Multiplikation, nicht aber die Subtraktion. Ist die Operation *assoziativ*, bedeutet dies, dass eine Verknüpfung aus zwei oder mehr Operationen stets zum selben Ergebnis führt – unabhängig von der Reihenfolge, in der die einzelnen Operationen ausgerechnet werden: `(a+b)+c=a+(b+c)`.

Sowohl für `Parallel.For` als auch für `Parallel.ForEach` gibt es Überladungen, die zusätzliche Parameter für die Reduktion definieren: den `Func`-Delegaten `localInit` und den `Action`-Delegaten `localFinally`. Aufgabe der Methode `localInit` ist es, die private threadlokale Variable zu initialisieren. Die Reduktion wird in der Methode `localFinally` durchgeführt.

Die Methode `localFinally` wird aufgerufen, nachdem die parallele Operation beendet wurde. Ihre Parameter sind das aktuelle Array-Element, der Schleifenstatus und die threadlokale Variable. Nur hier, in dieser Funktion, müssen Sie den Zugriff auf die threadlokale Variable synchronisieren – beispielsweise mithilfe einer `Monitor`-Klasse, einer `Interlocked`-Klasse oder der `lock`-Anweisung.

Die Basisüberladung der `Parallel.For`-Methode für Reduktion sieht wie folgt aus:

```
public static ParallelLoopResult For<TLocal>(
   int fromInclusive,
   int toExclusive,
```

```
    Func<TLocal> localInit,
    Func<int, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally
)
```

Und so sieht die Überladung der `Parallel.ForEach`-Methode aus:

```
public static ParallelLoopResult ForEach<TSource, TLocal>(
    IEnumerable<TSource> source,
    Func<TLocal> localInit,
    Func<TSource, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localFinally
)
```

In der folgenden Übung werden Sie nachzählen, wie viele Werte in einem Array größer als 5 sind – und so die Array-Werte zu einem einzigen Ergebniswert reduzieren.

Reduzieren Sie eine Datenmenge auf einen Zählwert

1. Legen Sie mit Visual Studio 2010 eine neue C#-Konsolenanwendung an. Fügen Sie eine `using`-Anweisung für den Namespace `System.Threading.Tasks` hinzu. Definieren Sie innerhalb der Klasse ein statisches Array mit den ganzen Zahlen 1, 10, 4, 3, 10, 20, 30 und 5 sowie eine `int`-Variable für den Zählwert.

   ```
   static int[] intArray = new int [] { 1, 10, 4, 3, 10, 20, 30, 5 };
   static int count=0;
   ```

2. Iterieren Sie in der `Main`-Methode mithilfe der Methode `Parallel.For` über das `int`-Array. In der Methode `localInit` initialisieren Sie die threadlokale Variable mit dem Wert 0. Für den `localFinally`-Delegaten, den letzten Parameter der `Parallel.For`-Methode, müssen Sie die drei Eingabeparameter für aktuelles Array-Element, Schleifenstatus und threadlokale Variable definieren.

   ```
   Parallel.For(0, intArray.Length, ()=>0, (index, loopState, subtotal)
   ```

3. In dem Lambda-Ausdruck für die Schleifenoperation prüfen Sie zunächst den Wert des aktuellen Array-Elements. Ist der Wert größer als 5, inkrementieren Sie den Zähler. Zum Schluss geben Sie die Thread-ID, den Index, den aktuellen Wert und das Teilergebnis aus. Liefern Sie das Teilergebnis zurück, damit es von der nächsten Iteration auf diesem Thread genutzt werden kann.

   ```
   if (intArray[index] > 5)
   {
       ++subtotal;
       Console.WriteLine("Thread {0}: Aufgabe {1}: Wert {2}, Teilergebnis {3}",
       Thread.CurrentThread.ManagedThreadId, index,
       intArray[index], subtotal);
   }
   return subtotal;
   ```

4. Nachdem die parallele Schleife abgearbeitet wurde, wird die Methode `localFinally` aufgerufen. Mithilfe der Methode `Interlocked.Add` können Sie hier auf threadsichere Weise die Teilergebnisse zusammenführen und das Gesamtergebnis berechnen.

   ```
   Interlocked.Add(ref count, subtotal);
   ```

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)

5. Zum Schluss geben Sie das Ergebnis der Zählung aus.

```
Console.WriteLine("Ergebnis der Zählung {0}", count);
```

Hier noch einmal der vollständige Quellcode der Anwendung:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Threading;

namespace Zaehlen
{
    class Program
    {
        static int[] intArray = new int[] { 1, 10, 4, 3, 10, 20, 30, 5 };
        static int count = 0;

        static void Main(string[] args)
        {
            Parallel.For(0, intArray.Length, () => 0, (index, loopState, subtotal) =>
                {
                    if (intArray[index] > 5)
                    {
                        ++subtotal;
                        Console.WriteLine("Thread {0}: Aufgabe {1}: Wert {2}, "
                                        + "Teilergebnis {3}",
                                        Thread.CurrentThread.ManagedThreadId, index,
                                        intArray[index], subtotal);
                    }
                    return subtotal;
                },
                (subtotal) =>
                {
                    Interlocked.Add(ref count, subtotal);
                });

            Console.WriteLine("Ergebnis der Zählung: {0}\n", count);
        }
    }
}
```

Listing 3.4 *Program.cs* aus dem Projekt *Zaehlen.csproj*

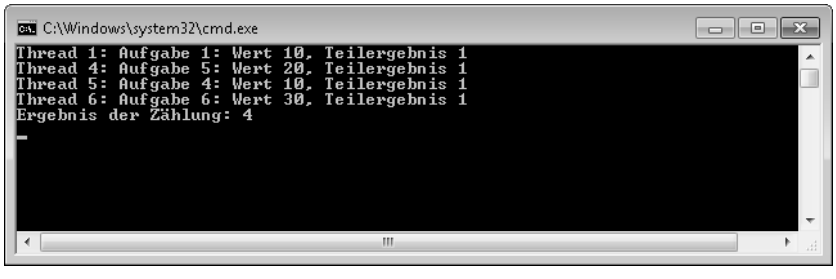

Abbildung 3.10 Hier wurde jedes Teilergebnis auf einem anderen Thread berechnet. Aus diesem Grund gibt es vier Teilergebnisse, die zu einem Gesamtergebnis reduziert werden.

HINWEIS Wie für die meisten Beispiele in diesem Buch gilt auch hier, dass die angezeigten Ergebnisse von der Anzahl verfügbarer Prozessorkerne abhängen.

In der nächsten Übung werden Sie die Methode `Parallel.ForEach` dazu benutzen, Fakultäten zu berechnen. Die Fakultät einer Zahl n ist das Produkt aller Zahlen von 1 bis n. Zum Beispiel ist die Fakultät von 5, geschrieben als 5!, gleich dem Produkt 5×4×3×2×1, also 120. Wie in dem vorangehenden Beispiel speichert die gemeinsam genutzte Variable das endgültige Ergebnis.

Reduzieren Sie eine Sammlung von ganzen Zahlen zu einer Folge von Fakultäten

1. Legen Sie mit Visual Studio 2010 eine neue C#-Konsolenanwendung an. Fügen Sie eine `using`-Anweisung für den Namespace `System.Threading.Tasks` hinzu. Definieren Sie über der `Main`-Methode ein statisches `int`-Feld `total` und initialisieren Sie es mit dem Wert 1. Definieren Sie auch noch eine Konstante `EXCLUSIVE` mit dem Wert 1. (Sie werden sie weiter unten dazu benutzen, die obere Schleifengrenze so zu korrigieren, dass der Maximalwert erreicht wird.) Definieren Sie schließlich noch ein einfaches Objekt, das Sie später im Programm als Sperre benutzen können.

```
static int total = 1;
const int EXCLUSIVE = 1;
static object mylock = new object();
```

2. Zur Berechnung der Fakultät von 5 setzen Sie einen `Parallel.ForEach`-Aufruf auf, der bei 1 startet und bei `(5+EXCLUSIVE)` endet. Dies ist der Bereich der zu multiplizierenden Zahlen. Initialisieren Sie das Teilergebnis mit dem Wert 1.

```
Parallel.For(1, 5+EXCLUSIVE, () => 1, (value, loopState, accumulator) =>
```

3. In der parallelen Operation multiplizieren Sie den Akkumulator (das Teilergebnis) mit dem Eingabewert und liefern das Ergebnis zurück.

```
accumulator*=value;
return accumulator;
```

4. Definieren Sie einen Lambda-Ausdruck für den `lastFinally`-Delegaten, mit dem Akkumulator als einzigem Eingabeparameter. In dem Lambda-Ausdruck richten Sie eine Sperre ein, die den Zugriff auf die gemeinsam genutzte Variable schützt. In dem geschützten Block berechnen Sie das Produkt aus dem Teilergebnis und dem aktuellen Wert von `total`.

```
lock (mylock)
{
   total *= accumulator;
}
```

5. Geben Sie das Ergebnis aus.

```
Console.WriteLine("Das Ergebnis lautet: {0}", total);
```

Hier noch einmal der vollständige Quellcode der Anwendung:

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace Fakultaet
{
    class Program
    {
        static int total = 1;
        const int EXCLUSIVE = 1;
        static object mylock = new object();
        static void Main(string[] args)
        {
            Parallel.For(1, 5 + EXCLUSIVE, () => 1, (value, loopState, accumulator) =>
            {
                accumulator *= value;
                return accumulator;
            },
            (accumulator) =>
            {

                lock (mylock)
                {
                    total *= accumulator;
                }
            });

            Console.WriteLine("Das Ergebnis lautet: {0}", total);
        }
    }
}
```

Listing 3.5 *Program.cs* aus dem Projekt *Fakultaet.csproj*

HINWEIS Die Beispiele in diesem Buch sind vor allem genau dies – Beispiele. Ihre wichtigste Aufgabe ist es, die im Buch besprochenen Themen oder Techniken zu veranschaulichen. Auf praxisnähere oder detailliertere Beispiele wurde daher weitgehend verzichtet.

Verwendung des MapReduce-Entwurfmusters

MapReduce ist ein bekanntes Entwurfsmuster, das im Jahr 2004 in dem Artikel »MapReduce: Simplified Data Processing on Large Clusters« von Jeffrey Dean und Sanjay Ghemawat eingeführt wurde. Wenn Sie möchten, können Sie sich den Text des Artikels von *http://labs.google.com/papers/mapreduce-osdi04.pdf* herunterladen. Ziel des MapReduce-Entwurfsmusters ist es, die Reduktion einer gewaltigen Datenmenge zu organisieren, die über mehrere Computer verteilt ist. Es ist aber auch im kleineren Maßstab anwendbar, beispielsweise auf modernen Multicore-Architekturen. Das MapReduce-Entwurfsmuster ist eine komplexe Kombination aus Datenparallelität, Abhängigkeiten und Reduktion.

Es gibt drei Datensammlungen in dem MapReduce-Entwurfsmuster. Die erste Datensammlung besteht aus den Eingabedaten für das MapReduce-Entwurfsmuster und hat die Form einer Auflistung von Schlüssel/Wert-Paaren. Auf dieser Auflistung führen Sie diverse Transformationen aus, woraufhin Sie als Zwischenergebnis die zweite Datensammlung erhalten – eine Auflistung von nicht eindeutigen Schlüssel/Wert-Paaren. Die dritte Datensammlung ist die Reduktion der zweiten Auflistung und ergibt sich durch Entfernung der nicht eindeutigen Schlüssel.

Zur Verdeutlichung sollten wir uns ein Beispiel ansehen. Ein ausgezeichnetes Beispiel für die Anwendung des MapReduce-Entwurfsmusters findet sich z.B. in dem Buch »Patterns for Parallel Programming: Understanding and Applying Parallel Patterns with the .NET Framework 4« von Colin Campbell, Ralph Johnson, Ade Miller und Stephen Toub, verfügbar unter *http://msdn.microsoft.com/en-us/library/ff963553.aspx*. Dort wird das MapReduce-Entwurfsmuster zur Zählung von Worthäufigkeiten über mehrere Dokumente verwendet. Die Dateien repräsentieren in diesem Fall die Eingabemenge, mit den Dateinamen als Schlüssel und den Speicherorten als Werte. Aus dieser ersten Auflistung wird dann die zweite Auflistung erzeugt – eine Liste von Wörtern und ihren Häufigkeiten. Taucht z.B. das Wort *Apfel* in drei der Eingabedateien auf, gibt es in der Zwischenauflistung drei Einträge für `Apfel`. Das Wort *Apfel* ist dann der Schlüssel und der Wert eines Schlüssels ist die Anzahl der Vorkommen in einer einzelnen Datei. Im abschließenden Reduktionsschritt werden die Schlüssel/Wert-Paare mit gleichlautendem Schlüssel zu einem Schlüssel/Wert-Paar zusammengefasst (siehe Abbildung 3.11).

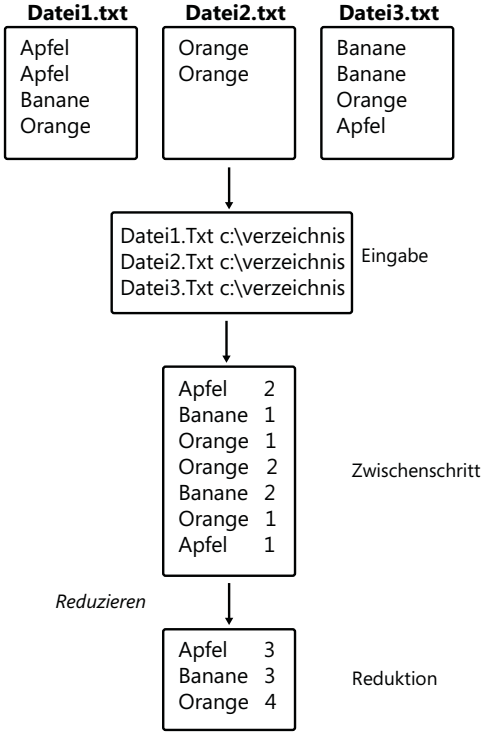

Abbildung 3.11 Anwendung des MapReduce-Entwurfsmusters zur Bestimmung von Worthäufigkeiten

Zur Implementierung des MapReduce-Entwurfsmusters bedarf es mehrerer Iterationen und Ebenen der Datenparallelität. Die Parallel Language Integrated Query (PLINQ) stellt eine Implementierung des MapReduce-Entwurfsmusters zur Verfügung (siehe nächstes Kapitel). Da es aber darüber hinaus in der TPL keine weitere Implementierung des MapReduce-Entwurfsmusters gibt, verwendet dieses Kapitel eine eigene Implementierung, gekapselt in die Klasse `MapReduce`.

HINWEIS Die `MapReduce`-Bibliothek ist in dem Begleitmaterial zu diesem Buch enthalten, siehe Hinweise in der Einführung des Buchs.

Die `MapReduce`-Klasse ist in dem Namespace `ParallelBook` definiert. Wenn Sie ein `MapReduce`-Objekt erzeugen, initialisieren Sie es mit einer Quellauflistung. Die `MapReduce`-Klasse enthält nur zwei Methoden. Die erste Methode, `MapReduce.Map`, ist dafür verantwortlich, aus der Quellauflistung die Auflistung mit dem Zwischenergebnis aufzubauen. Der erste Parameter ist die Abbildungsfunktion, die die Transformation durchführt. Der zweite und letzte Parameter ist ein `out`-Parameter – für die zweite Auflistung. Die zweite Methode heißt `MapReduce.Reduce`. Sie übernimmt als erstes Argument die Auflistung mit dem Zwischenergebnis und reduziert diese mithilfe der Reduktionsoperation, die als zweites Argument übergeben wird. (Die Operationen für die Abbildung und die Reduktion tauchen in der Schnittstelle der Klasse als eigenständige Methoden auf, damit es möglich ist, mehrere Reduktionen auf einer Zwischenauflistung auszuführen.) Der letzte Parameter ist die Gruppierungsoperation, die die Schlüssel in der Zwischenauflistung in Gruppen zusammenfasst. Die Gruppierung ist sehr wichtig, da die Zwischenauflistung anhand der Gruppen reduziert wird. Voreinstellung ist die Reduktion nach übereinstimmenden Schlüsseln, d.h., Elemente mit gleichem Schlüssel bilden eine Gruppe und werden zu einem Wert reduziert.

Der Prototyp der Methode `MapReduce.Map` sieht wie folgt aus:

```
public void Map<KEY2, VALUE2>(Func<Tuple<KEY, VALUE>,
   IEnumerable<Tuple<KEY2, VALUE2>>> mapFunc,
   out IEnumerable<Tuple<KEY2, VALUE2>> TupleCollection
)
```

Und hier der Prototyp für die Methode `MapReduce.Reduce`.

```
public IEnumerable<Tuple<KEY2, VALUE2>> Reduce<KEY2, VALUE2>(
   IEnumerable<Tuple<KEY2, VALUE2>> intermediate,
   Func<KEY2, VALUE2[], VALUE2> reduceFunc,
   Func<IEnumerable<Tuple<KEY2, VALUE2>>, Dictionary<KEY2, VALUE2[]>>
   groupFunc= null
)
```

Die nächste Übung demonstriert, wie die `MapReduce`-Klasse verwendet wird. Zunächst werden Sie eine Auflistung von Schlüssel/Wert-Paaren anlegen. Die Schlüssel sind Strings, die Werte ganze Zahlen (Integer). Das Zwischenergebnis ist eine Auflistung mit den quadrierten Werten aus der Eingabe-Auflistung. Im Reduktionsschritt werden die Schlüssel durch Aufaddierung reduziert.

Erzeugen Sie eine Instanz von MapReduce, mit der Sie die Werte in einer Quellauflistung quadrieren und anschließend durch Aufaddierung der Schlüssel reduzieren

1. Legen Sie mit Visual Studio 2010 eine neue C#-Konsolenanwendung an. Fügen Sie eine `using`-Anweisung für den Namespace `System.Threading.Tasks` hinzu. Richten Sie einen Verweis auf *MapReduce.dll* ein.

2. Definieren und initialisieren Sie in der `Main`-Methode ein Array von binären Tupeln für Paare aus einem String und einer ganzen Zahl.

```
Tuple<string, int>[] tuples = new Tuple<string, int>[] {
   new Tuple<string, int>("a", 3),
   new Tuple<string, int>("b", 2),
   new Tuple<string, int>("b", 5)
};
```

3. Erzeugen Sie eine Instanz der Klasse `MapReduce`. Im Konstruktor initialisieren Sie das Objekt mit dem Tupel-Array.

```
MapReduce<string, int> letters = new MapReduce<string, int>(tuples);
```

4. Nun zur Transformation der Quellauflistung. Definieren Sie zuerst eine Auflistung von Tupeln, in der das Zwischenergebnis gespeichert werden kann. Diese Auflistung übergeben Sie als zweites Argument an den `out`-Parameter der Methode `MapReduce.Map`. Als erstes Argument übergeben Sie Ihre Abbildungsoperation, die einfach den Wert eines einzelnen Tupels quadriert.

```
IEnumerable<Tuple<string, int>> newmap;
letters.Map<string, int>((input) =>
{
   return new Tuple<string, int>[] { new Tuple<string, int>(input.Item1,
      input.Item2 * input.Item2) };
}, out newmap);
```

5. Reduzieren Sie die Auflistung mithilfe der Methode `MapReduce.Reduce`. Übergeben Sie als Eingabe die Auflistung mit dem Zwischenergebnis und berechnen Sie für jede Gruppe die Gesamtsumme.

```
IEnumerable<Tuple<string, int>> reduction =
letters.Reduce<string, int>(newmap, (key, values) =>
{
   int total = 0;
   foreach (var item in values)
   {
      total += item;
   }
   return total;
});
```

6. Geben Sie das Ergebnis – also die von der Methode `MapReduce.Reduce` zurückgelieferte Auflistung – auf die Konsole aus. Es sollte für `a` die Summe 9 und für `b` die Summe 29 angezeigt werden.

```
foreach (var item in reduction)
{
   Console.WriteLine("{0} = {1}", item.Item1, item.Item2);
}
```

Hier noch einmal der vollständige Quellcode der Anwendung:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Buchstaben
{
        class Program
        {
            static void Main(string[] args)
            {
                Tuple<string, int>[] tuples = new Tuple<string, int>[] {
                   new Tuple<string, int>("a", 3),
                   new Tuple<string, int>("b", 2),
                   new Tuple<string, int>("b", 5) };

                MapReduce<string, int> letters = new MapReduce<string, int>(tuples);
                IEnumerable<Tuple<string, int>> newmap;

                letters.Map<string, int>((input) =>
                {
                    return new Tuple<string, int>[] { new Tuple<string,
                       int>(input.Item1, input.Item2 * input.Item2) };
                }, out newmap);

                IEnumerable<Tuple<string, int>> reduction = letters.Reduce<string,
                   int>(newmap, (key, values) =>
                {
                    int total = 0;
                    foreach (var item in values)
                    {
                        total += item;
                    }
                    return total;
                });

                foreach (var item in reduction)
                {
                    Console.WriteLine("{0} = {1}", item.Item1, item.Item2);
                }
            }
        }
}
```

Listing 3.6 *Program.cs* aus dem Projekt *Buchstaben.csproj*

Beispiel: Worthäufigkeiten

»There are more things in heaven and earth, Horatio, than are dreamt of in your philosophy.«[1]

– William Shakespeare

Shakespeare ist zeitlos. Sein Werk ebenso bedeutend wie wortgewaltig. Bestes Ausgangsmaterial also für eine Bestimmung der Worthäufigkeiten – und eine etwas praxisbezogenere Demonstration zur Verwendung der Klasse `MapReduce`.

[1] »Es gibt mehr Ding' im Himmel und auf Erden, als Eure Schulweisheit sicht träumt, Horatio.«

Das Beispiel analysiert vier bekannte Shakespearsche Sonette. Die Texte diese Sonette können von vielen Sites im Web heruntergeladen werden. Ziel der Analyse ist es, herauszufinden, wie oft die einzelnen Wörter in den vier Sonetten verwendet werden. Um zu verhindern, dass das Ergebnis zu sehr von kleinen Wörtern aus dem Grundwortschatz – wie »a«, »be«, »we« und so weiter – beherrscht wird, schließen wir kleine Wörter ganz aus. Glücklicherweise gibt es von der Methode `MapReduce.Map` eine Überladung, die genau für solche Fälle vorgesehen ist. Sie besitzt einen zusätzlichen `Filter`-Parameter, einen Funktionsdelegaten. Die Methode, die Sie diesem Parameter übergeben, muss als Argument ein Schlüssel/Wert-Paar übernehmen und `true` oder `false` zurückliefern, je nachdem, ob das Schlüssel/Wert-Paar in die Auflistung für das Zwischenergebnis aufgenommen oder übersprungen werden soll.

Die Quellauflistung enthält die Namen und Speicherorte der vier Sonette und wird zur Erzeugung einer Instanz der Klasse `MapReduce` verwendet.

```
Tuple<string, string>[] sonnets = new Tuple<string, string>[] {
new Tuple<string, string>("Sonnet 1.txt",@"C:\shakespeare"),
new Tuple<string, string>("Sonnet 2.txt",@"C:\shakespeare"),
new Tuple<string, string>("Sonnet 3.txt",@"C:\shakespeare"),
new Tuple<string, string>("Sonnet 4.txt",@"C:\shakespeare") };
MapReduce<string, string> wordCount = new MapReduce<string, string>(sonnets);
```

Mithilfe der Methode `MapReduce.Map` sollen die Dateinamen auf Worthäufigkeiten abgebildet werden.

1. Lesen Sie den Text der Sonette.
2. Definieren Sie, an welchen Trennzeichen ein Wortende erkannt werden kann.
3. Erzeugen Sie ein Wörterbuch (in Form eines `Dictionary`-Objekts). Prüfen Sie für jedes Wort, ob es bereits im Wörterbuch enthalten ist. Falls nicht, fügen Sie das Wort ein und setzen Sie seinen Zähler auf 1. Andernfalls, also wenn das Wort bereits eingetragen wurde, inkrementieren Sie den Zähler für den Worteintrag. Am Ende der Operation liefern Sie den `Values`-Anteil des `Dictionary`-Objekts als Ergebnis zurück – dies ist die Auflistung mit dem Zwischenergebnis, in der die Worthäufigkeiten für alle vier Eingabedateien enthalten sind (d.h., ein Wort kann bis zu viermal in der Auflistung auftreten).

Der zugehörige Code sieht damit bisher wie folgt aus:

```
IEnumerable<Tuple<string, int>> wordCollection;
wordCount.Map<string, int>((input) =>
{
   StreamReader sw = new StreamReader(input.Item2 + @"\" + input.Item1);
   string data = sw.ReadToEnd();
   string[] words = data.Split(new[] {' ','.',',',';',':','=','+', '-', '*', ')',
                                      '(', '!', '#', '$', '\n', '\r'});
   Dictionary<string, Tuple<string, int>> rawCount =
                                     new Dictionary<string Tuple<string, int>>();
   foreach (var word in words)
   {
      Tuple<string, int> value;
      if (rawCount.TryGetValue(word, out value))
      {
         int increment = rawCount[word].Item2 + 1;
         rawCount[word] = new Tuple<string, int>(word, increment);
      }
      else
      {
         rawCount.Add(word, new Tuple<string, int>(word, 1));
      }
   }
   return rawCount.Values;
},
```

Nach der Abbildungsfunktion definieren Sie die Filterfunktion. Um die Auflistung nicht zu sehr ausufern zu lassen, werden Wörter mit weniger als drei Zeichen ausgeschlossen:

```
(key, value) =>
{
   if (key.Length < 3)
   {
      return false;
   }
   else
   {
      return true;
   }
},
out wordCollection);
```

Der Aufruf der Methode `MapReduce.Reduce` ist vergleichsweise einfach. Die übergebene Reduktionsoperation reduziert die Schlüssel-Gruppen zu Gesamtsummen, die angeben, wie oft ein Wort in allen vier Dateien zusammen vorkommt.

```
IEnumerable<Tuple<string, int>> reduction = wordCount.Reduce(
   wordCollection,
   (key, values) =>
   {
      return values.Sum();
   }
);
```

Zum guten Schluss geben Sie die Ergebnisse aus.

```
foreach (var item in reduction)
{
   Console.WriteLine("{0} {1}", item.Item1, item.Item2);
}
```

Abbildung 3.12 Auszug aus der Ausgabe des Worthäufigkeiten-Beispiels

Zusammenfassung

Datenparallelität ist die Anwendung einer parallelen Operation auf eine Datensammlung und wird von der TPL durch die Methoden `Parallel.For` und `Parallel.ForEach` unterstützt. Sofern es keine Abhängigkeiten gibt, ist die Umwandlung einer sequenziellen `for`-Schleife in eine parallele Schleife relativ simpel. Die Blockgröße für die parallele Operation wird dabei vom Standardpartitionierer der TPL festgelegt.

Um eine parallele Schleife abzubrechen, rufen Sie eine der Methoden `ParallelLoopState.Break` oder `ParallelLoopState.Stop` auf. Die Methode `ParallelLoopState.Stop` führt schneller zur vollständigen Beendigung der Schleife; doch selbst bei einem `Stop`-Abbruch kann es Schleifeniterationen geben, die trotz des Abbruchs ungestört zu Ende ausgeführt werden. Länger laufende Aufgaben sollten allerdings regelmäßig überprüfen, ob ein Abbruch gewünscht wird.

Unbehandelte Ausnahmen aus parallelen Schleifen werden im Verbindungsthread abgefangen und behandelt. Da in parallel ausgeführten Aufgaben mehrere unbehandelte Ausnahmen gleichzeitig auftreten können, werden unbehandelte Ausnahmen stets in Form einer »Sammel«-Ausnahme vom Typ `AggregateException` an den Verbindungsthread weitergeleitet. Über die `AggregateException.InnerExceptions`-Auflistung können Sie auf die einzelnen unbehandelten Ausnahmen zugreifen. Nachdem eine unbehandelte Ausnahme ausgelöst wurde, enthält die Eigenschaft `ParallelLoopState.IsExceptional` den Wert `true`. Laufende Aufgaben können anhand dieses Werts prüfen, ob eine Ausnahme ausgelöst wurde und sich bei nächster Gelegenheit selbst beenden.

Reduktion bedeutet, eine Menge von Werten zu einem Ergebniswert zu reduzieren. Sowohl für die Methode `Parallel.For` als auch für `Parallel.ForEach` gibt es Überladungen, die die Reduktion unterstützen. Der Effekt von Iterationen, die auf dem gleichen Thread ausgeführt werden, wird dabei direkt zu Teilergebnissen zusammengefasst. Da dabei mit einer privaten threadlokalen Variable gearbeitet werden kann, bedarf es keiner Synchronisierung. Wenn alle Teilergebnisse vorliegen, wird die `lastFinally`-Operation aufgerufen, um die Teilergebnisse zum endgültigen Gesamtergebnis zusammenzufassen.

Das MapReduce-Entwurfsmuster dient zur Reduktion von sehr großen Datenmengen, die über mehrere Server verteilt sind, kann aber auch in der Umgebung eines einzelnen Multicore-Computers verwendet werden. In PLINQ ist eine Implementierung des MapReduce-Entwurfsmusters enthalten. Eine weitere Implementierung, in Form der Klasse `MapReduce`, können Sie zusammen mit dem Begleitmaterial zu diesem Buch herunterladen.

Schnellreferenz

Um dies zu erreichen	Tue Folgendes
Mithilfe paralleler Aufgaben eine Auflistung durchlaufen	Verwenden Sie die Methode `Parallel.For`
Mithilfe paralleler Aufgaben direkt über die Elemente einer Auflistung iterieren	Verwenden Sie die Methode `Parallel.ForEach`
Eine `Parallel.For`-Schleife abbrechen	Verwenden Sie `ParallelLoopState.Break` oder `ParallelLoopState.Stop` zum kooperativen Abbruch. `ParallelLoopState.Stop` ist das klarere Modell und bricht sowohl aktuelle als auch zukünftige Aufgaben ab
Ausnahmen aus parallelen Aufgaben behandeln	Fangen Sie im Verbindungsthread die `AggregateException`-Ausnahme ab. Die Originalausnahme finden Sie in der Eigenschaft `AggregateException.InnerException`. Wenn mehr als eine Ausnahme ausgelöst wurde, iterieren Sie über die `AggregateException.InnerExceptions`-Auflistung.
Eine Reduktion, beispielsweise eine Summenbildung, durchführen	Nutzen Sie die `localInit`- und `localFinally`-Parameter der `Parallel.For`- bzw. `Parallel.ForEach`-Methoden

Donis Marshall: Parallele Programmierung mit Visual Studio 2010 - Crashkurs, Microsoft Press 2011 (ISBN 978-3-86645-555-9)