Kurzreferenz („Cheat Sheet“) Task Parallel Library (TPL)

Autor: Bernd Marquardt (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

Anwendung

Die **Task Parallel Library (TPL)** kann ab .NET Framework 4.0 und auch in .NET Core benutzt werden. Die Bibliothek ermöglicht eine einfache und übersichtliche Parallelisierung von managed Code (C#, VB.NET u.a.).

- Parallele Datenabfragen mit PLINQ
- Paralleles Ausführen von Schleifen
- Paralleles Ausführen von Code-Bereichen
- Paralleles Ausführen von Aufgaben (Tasks)
- Bereitstellung von Container-Klassen, die im parallelen Umfeld eingesetzt werden können
- Bereitstellung von Synchronisierungselementen
- Möglichkeiten für die Behandlung von Laufzeitfehlern
- Abbrechen von nebenläufigen Operationen

Bibliothek und Namensräume

Alle Klassen der **TPL** sind in der **System.dll** des .NET Frameworks (ab 4.0) enthalten.
Es werden die Namensräume *System.Threading.Tasks* und *System.Threading* benötigt. Bei der Nutzung von PLINQ ist der Namensraum *System.Linq* ebenfalls erforderlich.

Parallele Datenabfragen mit PLINQ

Die parallele Ausführung eines LINQ-Befehls wird durch die Erweiterungsmethode *.AsParallel()* ermöglicht. Um eine deutliche Performance-Steigerung zu sehen, muss die verarbeitete Datenmenge ausreichend groß sein!

```
string[] arrNames = { "Bernd", "Ute", "Monika", "Willi", "Alina" };
var q = from x in arrNames.AsParallel()
        where x.Length == 5
        orderby x
        select x;
```

Die Anzahl der zu benutzenden Threads kann ebenfalls mit einer Erweiterungsmethode angegeben werden:

```
arrNames.AsParallel().WithDegreeOfParallelism(4)
```

Paralleles Ausführen von Schleifen

Schleifen können dann parallel verarbeitet werden, wenn die einzelnen Schleifendurchläufe unabhängig voneinander sind, d.h., sie können in beliebiger Reihenfolge ausgeführt werden. Außerdem dürfen nicht mehrere Threads schreibend auf die gleiche Variable zugreifen. Hier wäre zusätzlich eine Synchronisierung erforderlich.
Die Parallelisierung wird mit der *Parallel*-Klasse programmiert. Der Körper der Schleife wird am einfachsten mit Hilfe einer Lambda-Funktion implementiert, kann aber auch mit Delegaten definiert werden.

```
double[] arrData2 = new double[10000];
Parallel.For(0, 10000, i =>
{
  arrData2[i] = Math.Sqrt(i) + Math.Sin((double)i / 1000.0);
});
```

Die *For()*-Methode in der *Parallel*-Klasse ist mehrfach überladen, um unterschiedliche Schleifen-Szenarien abzubilden. Die Schrittweite einer *Parallel.For()*-Schleife muss jedoch immer +1 sein. Ggf. muss der Schleifenindex entsprechend umgerechnet werden:

```
double[] arrData2 = new double[10000];
Parallel.For(0, 10000, i =>
{
  double x = (double)(i + 1) * 0.5;
  arrData2[i] = Math.Sqrt(x);
});
```

Variablen, die innerhalb einer Schleife deklariert werden, sind „thread-lokal“, d.h., in jedem Thread steht eine eigene Instanz der Variablen zur Verfügung. Eine Synchronisierung ist in diesem Fall nicht erforderlich.
Beachten Sie bitte, dass am Ende einer *Parallel.For()*-Schleife synchronisiert wird. D.h., der nachfolgende Code wird erst dann ausgeführt, wenn alle Threads mit der Schleifenverarbeitung komplett fertig sind. *Parallel.For()* ist also ein „synchroner“ Aufruf, der jedoch innerhalb des Aufrufs parallel arbeitet.
Foreach-Schleifen können ebenfalls mit der *Parallel*-Klasse parallelisiert werden. Auch hier müssen die Schleifendurchläufe unabhängig voneinander sein, da die einzelnen Elemente in beliebiger Reihenfolge verarbeitet werden.

```
string[] names = new string[] { "Bernd", "Hans", "Willi", "Egon", "Gerd" };
Parallel.ForEach(names, item =>
{
   Console.WriteLine("{0}\t hat {1} Zeichen.", item, item.Length);
});
```

Wichtig: Vermeiden Sie **Data Races**! Data Races sind Code-Stellen, an denen z.B. mehrere Threads schreibend auf eine Variablen-Instanz zugreifen. Der folgende Code liefert also ein falsches Ergebnis:

```
double dSum = 0.0;
Parallel.For(0, 1000, i =>
{
  dSum += Math.Sqrt(i); // ACHTUNG: Data Race!!!
  // Liefert eine falsches Ergebnis!!!
});
```

Wichtig: Vermeiden Sie Schleifen, deren Durchläufe abhängig voneinander sind. Der folgende Code liefert ebenfalls ein falsches Ergebnis:

```
double[] arrData2 = new double[n];
arrData2[0] = 1.0;
Parallel.For(1, n, i =>
{
   arrData2[i] = arrData2[i - 1] + 1.0;   // FEHLER!!!
});
```

Abhängigkeiten in Schleifen lassen sich in vielen Fällen durch einfaches Umstellen des Codes eliminieren.
Aggregationen (oder Reduktionen) werden benötigt, wenn z.B. viele Zahlen addiert werden müssen. Wie im vorletzten Beispiel gezeigt wurde, muss hier besonders auf sog. „Data Races“ geachtet werden. Für die Aggregation wird eine Überladung der *Parallel.For*-Methode genutzt, welche eine interne Variable, den sog. „thread-lokalen Status“ anwendet:

```
static object aggLock = new object();
static void Main(string[] args)
{
  // Serieller Test
  double dTest = 0.0;
  for (int i = 0; i < 100000000; i++)
  {
    dTest += Math.Sqrt(i);
  }
  // Paralleler Test
  double dSum = 0.0;
  Parallel.For(0, 100000000,
    () => 0.0,            // Initialisierung
    (i, pls, tls) =>      // Teilsumme
    {
      tls += Math.Sqrt(i);
      return tls;
    },
    (partSum) =>      // Gesamtsumme
    {
      lock (aggLock)
      {
        dSum += partSum;
      }
    });
  Console.WriteLine("Test: {0}   Parallel: {1}", dTest, dSum);
}
```

Es werden drei Lambda-Funktionen verwendet: Zunächst wird der „thread-lokale Zustand“ mit einer Lambda-Funktion initialisiert. Danach kommt die Lambda-Funktion für den Schleifenkörper, in der die Teilsummen berechnet werden. Schließlich werden diese Teilsummen in der dritten Lambda-Funktion addiert. Beachten Sie, dass ohne das *lock*-Statement wiederum ein Data Race vorgelegen hätte. Die Sperrungen in dieser Funktion treten aber sehr selten auf, so dass die Performance dadurch nicht verringert wird.

Hinweis: Wenn z.B. *Integer*-Zahlen in der dritten Lambda-Funktion verarbeitet werden sollen, kann man die sehr ressourcen-schonende *Interlocked*-Klasse einsetzen:

```
Interlocked.Add(ref iSum, partSum);
```

Schleifen können mit Hilfe der Lambda-Variablen vom Typ *ParallelLoopState* (*pls*) abgebrochen werden. Es werden alle Threads, welche an der Schleife parallel arbeiten, angehalten.
ParallelLoopState.Stop() hält alle Threads der Schleife schnellstmöglich an.
ParallelLoopState.Break() hält die Schleife ebenfalls an, jedoch werden bis zum aktuellen Haltepunkt alle Schleifendurchläufe ausgeführt. Dieser Aufruf entspricht also dem *break* in einer normalen *for*-Schleife.
Die Anzahl der zu verwendenden Threads in der parallelen Schleife kann mit einer Instanz der Klasse *ParallelOptions* gesteuert werden.

```
// Nur zwei Threads benutzen
var options = new ParallelOptions { MaxDegreeOfParallelism = 2 };

Parallel.For(0, 10000, options, i =>
{
  dRes[i] = Math.Sqrt(i);
});
```

Paralleles Ausführen von Code-Bereichen

Code-Bereiche können durch Aufruf von *Parallel.Invoke()* nebenläufig ausgeführt werden:

```
Parallel.Invoke( () =>
{
  for (int i = 0; i < 10000; i++)
  {
    dY1[i] = Math.Sqrt(dX[i]) + Math.Sin(dX[i]);
  }
}, () =>
{
  for (int i = 0; i < 10000; i++)
  {
```

Kurzreferenz („Cheat Sheet“) Task Parallel Library (TPL)

Autor: Bernd Marquardt (www.IT-Visions.de)

www.IT-Visions.de
Dr. Holger Schwichtenberg

```
  dY2[i] = Math.Sqrt(dX[i] / 2.0) + Math.Cos(dX[i]);
 }
});
```

Sie können beliebig viele Lambda-Funktionen bei einem *Invoke()*-Aufruf benutzen. Der *Invoke()*-Aufruf wird erst dann beendet, wenn alle Teilfunktionen im Aufruf komplett abgearbeitet worden sind. Somit ist *Parallel.Invoke()* auch ein synchroner, blockierender Aufruf.
In einem *Parallel.Invoke()*-Aufruf können auch Delegaten für die Funktionsübergabe benutzt werden.

Paralleles Ausführen von Aufgaben (Tasks)

Die *Task*-Klasse wird benötigt, um erweiterte Parallelisierungsprobleme leichter lösen zu können. Der .NET-Threadpool stellt dem *Task-Manager* der TPL eine bestimmte Anzahl von Threads zur Verfügung. Der *Task-Manager* kann diese nun nutzen, um die anstehenden Aufgaben nebenläufig abzuarbeiten.
Der *Task-Manager* verteilt die Aufgaben so, dass immer eine möglichst optimale Anzahl von Threads die Aufgaben nebenläufig verarbeitet. Hierbei kann es auch zum sog. **Task Stealing** kommen. In diesem Fall werden Aufgaben (Tasks) von einem Thread, der viel zu tun hat, zu einem anderen Thread verschoben, der nicht zu tun hat.

```
Task t1 = Task.Factory.StartNew( () =>  MethodA() );
Task t2 = Task.Factory.StartNew( () =>  MethodB() );
Console.WriteLine("Task 1: {0}\nTask 2: {1}", t1.IsCompleted, t2.IsCompleted);
t1.Wait();
t2.Wait();
Console.WriteLine("Task 1: {0}\nTask 2: {1}" t1.IsCompleted, t2.IsCompleted);
```

Hierbei ist zu beachten, dass der Aufruf von *Task.Factory.StartNew(…)* den auszuführenden Code **nicht** blockiert. Der neue Task wird dem Task-Manager übergeben, der ihn dann einem Thread zuordnet. Erst die beiden *Wait*-Methodenaufrufe blockieren den Code. Dort wird also gewartet, bis die beiden gestarteten Aufgaben beendet sind.

Wichtig: Es wird nicht für jeden Task ein Thread erzeugt, sondern alle Tasks werden auf vorhandene Threads verteilt! Die Anzahl der zu nutzenden Threads wird von Task-Manager zur Laufzeit ermittelt.
Hinweis: Tasks können sehr gut benutzt werden, um in die Benutzerschnitt-stellen die Blockierung von Ereignismethoden zu verhindern.

Tasks können nach ihrer Ausführung eine Fortsetzungsmethode aufrufen. Hierbei kann das Ergebnis des Tasks an die *ContinueWith*-Methode weitergegeben werden:

```
// Nebenläufige Berechnung der Quadratwurzeln
Task.Factory.StartNew(() =>
{
  // … Berechnung …
  return dErgebnis;
}).ContinueWith(dWert =>
{
 // Aufruf, wenn Berechnung beendet ist, Verarbeitung des Ergebnisses
});
```

Achten Sie bei der Parameterübergabe an Tasks auf mögliche Seiteneffekte. Im folgenden Beispiel wird der *string*-Parameter als Lambda-Parameter an die *printMessage*-Methode weitergegeben, ansonsten wird (höchstwahrscheinlich) einfach fünf Mal der letzte Name in der Liste („Markus“) ausgegeben. Eine andere Lösung besteht darin, explizit in jedem *foreach*-Durchlauf eine *string*-Variable zu deklarieren und die Schleifenvariable *s* hineinzukopieren.

```
static void Main(string[] args)
{
 string[] names = { "Bernd", "Alina", "Monika", "Willi", "Markus" };
 foreach (string s in names)
 {
  // RICHTIG!!! Benutzung eines Action-Delegates: Action<object>
  Task myTask = new Task((obj) => printMessage((string)obj), s);
  myTask.Start();
 }
 Console.WriteLine("Fertig. Eine Taste drücken.");
 Console.ReadLine();
}
static void printMessage(string message)
{
 Console.WriteLine("Message: {0}", message);
}
```

Tasks können beliebige Rückgabewerte (generisch) sehr leicht zurückgeben. Ein Task, der z.B. als Ergebnis einen doppeltgenauen Wert zurückgeben soll, wird folgendermaßen deklariert:

```
Task<double> task = Task.Factory.StartNew(() => CalcMethod(0, 50000));
```

Die Methode *CalcMethod()*, welche im *Task* laufen soll, muss nun so programmiert werden, dass sie einen doppeltgenauen Wert als *return*-Parameter enthält. Das Ergebnis der Berechnung kann über das Objekt *task* abgefragt werden:

```
double dResult = task.Result;
```

Wenn die nebenläufige Aufgabe beim Aufruf von *task.Result* bereits fertig ausgeführt ist, wird sofort das Ergebnis zurückgegeben und der Aufruf blockiert nicht. Ist die Aufgabe noch nicht komplett ausgeführt, blockiert der Code an dieser Stelle, bis das Ergebnis schließlich zur Verfügung steht.

Bereitstellung von Synchronisierungselementen

Neben den bereits bekannten Synchronisierungselementen (lock, Monitor, Semaphor, …) gibt es seit .NET 4.0 weitere, neue Elemente:

- *Barrier* – Warten bis n Threads eine Stelle passieren können
- *CountDownEvent* – Ereignis, wenn heruntergezählt wurde
- *SpinLock*
- *SpinWait*
- Thread-sichere *Collection*-Klassen (*Stack, Queue, Dictionary*)
- …

Behandlung von Laufzeitfehlern

Es gibt einen neue Ausnahme-Klasse: *AggregateException*
Die neue Ausnahme kann ganz normal mit den bekannten Ausnahmen verwendet werden. Die Idee ist, dass bei parallelem Code mehrere Ausnahmen „gleichzeitig“ auftreten können. Die Laufzeitumgebung sammelt alle im parallelen Code auftretenden Ausnahmen ein und wirft dann ein neues *AggragateException*-Objekt. In diesem Objekt können alle gesammelten Ausnahmen über die Eigenschaft *InnerExceptions* abgefragt und bearbeitet werden.

Abbrechen von nebenläufigen Operationen

Threads, *Tasks* oder parallele Schleifen dürfen nicht einfach „zerstört“ werden, wenn man sie beenden will. Bevor nebenläufige Objekte beendet werden, sollten sie ihre verwendeten Ressourcen ordnungsgemäß freigeben (Dateien schließen, Bitmaps freigeben, usw.). Diese Möglichkeit bietet des *Unified Cancellation Framework* in der TPL. Wenn eine nebenläufige Operation abgebrochen werden soll, erzeugt man ein *CancellationTokenSource*-Objekt. Diese kann *CancellationToken*-Objekte erzeugen, welche man in die Tasks oder Threads weitergibt, die eventuell abgebrochen werden sollen. Um die nebenläufigen Operationen zu beenden, ruft man die *Cancel()*-Methode der TokenSource auf. Im Code der nebenläufigen Operation wird in regelmäßigen Abständen mit der Eigenschaft *IsCancellationRequested* abgefragt, ob die Operation nun beendet werden soll.

```
static void Main(string[] args)
{
 CancellationTokenSource cts = new CancellationTokenSource();
 Task t = new Task(() => LongRunningTask(0, 200000000, cts.Token));
 t.Start();
 // Task abbrechen
 cts.Cancel();
 try {
  t.Wait();
 }
 catch (AggregateException ex)  { // …  }
}
private static double LongRunningTask(CancellationToken token)
{
 // …
 // Soll die Operation beendet werden???
 if (token.IsCancellationRequested)
 { // Hier alles aufräumen… dann:
  token.ThrowIfCancellationRequested();
 }
}
```

Links

Die Task Parallel Library (Microsoft MSDN)
http://msdn.microsoft.com/de-de/library/dd460717(v=vs.110).aspx
Task Parallel Library (Code Project)
http://www.codeproject.com/Articles/152765/Task-Parallel-Library-1-of-n
Parallel Extensions (Wikipedia)
http://de.wikipedia.org/wiki/Parallel_Extensions
What is TPL? (YouTube)
http://www.youtube.com/watch?v=No7QqSc5cI8
Overview of the Task Parallel Library (TPL)
http://architects.dzone.com/articles/overview-task-parallel-library
Task Parallel Library (Channel 9)
http://channel9.msdn.com/Tags/task+parallel+library

Über den Autor

Wenn es den Begriff "Urgestein" auch in der Microsoft-Welt gibt, dann trifft er auf **Bernd Marquardt** mit Sicherheit zu. Er programmiert seit 1975 und schon seit 1987 mit Windows. Lange Jahre er sich mit den Tiefen der COM-Programmierung unter Visual C++ und Visual Basic beschäftigt. Heute ist einer der bekanntesten deutschen Experten für .NET und C#, insbesondere den Oberflächentechnologien, Multi-Threading, Parallelprogrammierung sowie Fragen der Interoperabilität zwischen C++ und C#.
E-Mail: B.Marquardt@IT-Visions.de
Modifikationen von Dr. Holger Schwichtenberg in Version 1.1