

Kochen mit vielen Köchen

Die Task Parallel Library in .NET 4.0 vereinfacht die Parallelprogrammierung. Wo der Entwickler bisher mühsam mit Threads gearbeitet hat, kann er nun auf fertige Klassen zurückgreifen. Sie ermöglichen es, Schleifen und Codebereiche zu parallelisieren sowie mit Tasks und Abbruchsituationen ordentlich umzugehen.

Auf einen Blick



Bernd Marquardt ist selbstständiger Berater und freier Autor und hält Vorträge auf Fachkonferenzen. Seit 2004 ist er Microsoft Most Valuable Professional (MVP) für C++ in Deutschland. Seine Schwerpunkte liegen beim Programmieren grafischer und mathematischer Algorithmen, der Parallelprogrammierung und dem .NET Framework.

Inhalt

- Schleifen parallelisieren.
- Daten aggregieren.
- Codebereiche parallel ablaufen lassen.

Serie

1. Grundlagen der TPL
2. Die Klasse Task der TPL
3. Synchronisierung und Cancellation Framework



dnpCode
A1009TPL

Eine Parallelisierung von Code ist immer eine Herausforderung für den Softwareentwickler. Es gilt, einige Dinge zu beachten: Data Races, Synchronisierung, Deadlocks, Granularität, Amdahls Gesetz und vieles mehr. Dazu kommt der Umstand, dass es bereits mehrere Technologien für die Parallelisierung von Code gibt: OpenMP [1], MPI [2, 3], UPC, TBB, PPL [4] und, und, und ... Viele dieser Technologien sind jedoch mit dem .NET Framework, also mit verwaltetem Code, nicht einsetzbar.

Obwohl in der sogenannten „Business-Programmierung“ die Entwicklung parallelen Codes vielleicht nicht ganz so wichtig ist wie zum Beispiel in naturwissenschaftlichen Bereichen, muss sich auch der „Business-Entwickler“ darüber im Klaren sein, dass mittlerweile auf fast jedem Schreibtisch ein Computer steht, dessen Prozessor mehrere Kerne enthält. Die vielen Kerne dieser modernen Prozessoren kann man natürlich ausnutzen, indem man einfach mehrere Programme gleichzeitig laufen lässt. Da hilft das Betriebssystem, das die Ausführung dieser Anwendungen auf die Prozessorkerne verteilt. Will der Softwareentwickler jedoch alle Prozessorkerne in einer Anwendung ausnutzen, dann muss er etwas dafür tun. Bisher musste er mühsam Multithreading programmieren. Ab sofort jedoch können Entwickler die Task Parallel Library (TPL) aus dem .NET Framework 4 nutzen.

Auch hier geht es eigentlich wieder um die sogenannte „Wie-“ beziehungsweise „Was“-Programmierung. Während der Entwickler bei der Benutzung von Multithreading genau programmieren muss, *wie* das Problem parallel gelöst werden soll, gibt er bei der TPL eher an, *was* er haben will. Die Bibliothek übernimmt den Rest – und das ist nicht gerade wenig.

Trotzdem sei eine Warnung ausgesprochen: Auch mit der neuen Bibliothek TPL wird das Erstellen von parallelem Code nicht einfach! Der Entwickler muss nach wie vor viele Dinge im Auge behalten und die richtige Implementierung wählen. Allerdings nimmt ihm die TPL sehr viele Verwaltungs- und Randaufgaben ab, die bisher zusätzlich noch angefallen sind und deren Implementierung den Code oft sehr unleserlich gemacht hat. Dazu zählen unter anderem die

Thread-Erzeugung, die Synchronisierung, die Übergabe von Parametern und Ergebnissen und die Steuerung der Threads.

Schleifen parallelisieren

Am einfachsten lassen sich in der Regel normale *for*-Schleifen parallelisieren. Hierbei wird schlicht versucht, die einzelnen Schleifendurchläufe auf die vorhandenen Prozessorkerne zu verteilen. Ein simples Beispiel: Der Computer enthält einen

Listing 1

Eine Schleife parallel ausführen.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
namespace TPL_1_1
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] dWerte1 = new double[1000];
            // Normaler sequenzieller Code
            for (int i = 0; i < 1000; i++)
            {
                double x = (double)i;
                dWerte1[i] = Math.Sqrt(x) + 3.14 * x / 2.0;
            }
            double[] dWerte2 = new double[1000];
            // Paralleler Code mit TPL
            Parallel.For(0, 1000, i =>
            {
                double x = (double)i;
                dWerte2[i] = Math.Sqrt(x) + 3.14 * x / 2.0;
            });
            // Test-Code (wieder sequenziell)
            for (int i = 0; i < 1000; i++)
            {
                if (dWerte1[i] != dWerte2[i])
                {
                    Console.WriteLine("Fehler bei Wert#{0}", i);
                }
            }
        }
    }
}
```

Prozessor mit zwei Kernen, und die *for*-Schleife soll von 1 bis 1000 laufen. In diesem Fall könnte der erste Kern die Schleifendurchläufe von 1 bis 500 ausführen und der zweite die Durchläufe von 501 bis 1000. Auf diese Weise sollte der Programmteil mit der Schleife deutlich schneller ausgeführt werden, da sich ja nun zwei Prozessorkerne um die Abarbeitung kümmern. Listing 1 zeigt den Code zu einem entsprechenden Beispiel, wenn die TPL eingesetzt wird.

In diesem Beispiel werden zunächst die beiden wichtigen Namensräume *System.Threading* und *System.Threading.Tasks* eingefügt. In diesen Namensräumen befinden sich alle Klassen, die mit der TPL etwas zu tun haben. Der erste Bereich in der *Main*-Methode zeigt eine ganz normale Schleife, welche mit dem C#-Befehl *for* erstellt wurde. In der Schleife wird ein bisschen gerechnet und die einzelnen Array-Elemente werden mit den berechneten Daten gefüllt.

Im zweiten Teil der *Main*-Methode wird die gleiche Schleife parallel ausgeführt. In diesem Fall wird die statische *For*-Methode aus der *Parallel*-Klasse der TPL verwendet, um die Berechnungen auszuführen. Die TPL stellt einen weiteren Thread bereit, einen gibt es ja schon, nämlich den Haupt-Thread. Beide Threads teilen sich dann die Verarbeitung der Schleife. Die Schleife läuft übrigens in diesem Fall von „inklusive“ 0 bis „exklusive“ 1000.

Interessant ist hier, wie der Schleifenrumpf definiert wird. Es kommt nämlich eine Lambda-Funktion zur Anwendung. Dieses Feature wurde in Version 3 von C# eingeführt. Der Platzhalter *i* stellt die Laufvariable der Schleife dar. Sie darf nicht explizit deklariert werden, denn sie wird von der *Parallel.For*-Methode zur Verfügung gestellt. Natürlich kann man auch einen anderen Namen wählen.

In diesem Fall ist das Vorgehen bei der Parallelisierung korrekt, denn obwohl zwei Threads gleichzeitig in ein Array Werte schreiben, werden immer unterschiedliche Array-Elemente angesprochen. Es liegt also keine sogenannte Data Race vor. Anders ausgedrückt: Zu keinem Zeitpunkt der Ausführung schreiben mehrere Threads gleichzeitig in ein Array-Element.

Der letzte Teil der Beispielanwendung dient lediglich der Überprüfung der Inhalte beider Arrays. Sind die Elemente unterschiedlich, so wird eine Fehlermeldung ausgegeben.

Aan dieser Stelle muss ein weiterer, entscheidender Punkt erwähnt werden. Die Schleife selbst wird zwar parallel auf meh-

Listing 2

Eine fehlerhafte Schleife.

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

namespace TPL_1_2
{
    class Program
    {
        static void Main(string[] args)
        {
            int nAnzahl = 1000;
            int[] iWerte = new int[nAnzahl];
            iWerte[0] = 1;
            // ACHTUNG: Fehlerhaftes Ergebnis!!
            // Schleifendurchläufe nicht
            // unabhängig
            Parallel.For(1, nAnzahl, i =>
            {
                iWerte[i] = iWerte[i - 1];
            });
            int iSum = 0;
            for (int i = 0; i < nAnzahl; i++)
            {
                iSum += iWerte[i];
            }
            Console.WriteLine("Ergebnis: {0}",
                iSum);
        }
    }
}
```

renen Prozessorkernen ausgeführt. Die Methode *Parallel.For* ist jedoch eine synchrone Methode. Das bedeutet, dass der nachfolgende Testcode im Beispiel erst dann ausgeführt wird, wenn alle Threads in der Methode *Parallel.For* ihre Arbeit beendet haben. Dieses Verhalten ist auch sehr sinnvoll, denn so kann man sicher sein, dass alle Werte im Array *dWerte2* berechnet sind, wenn das *Parallel.For*-Konstrukt verlassen wird.

Bei dieser Art der Parallelisierung von Schleifen heißt es jedoch aufpassen, denn es gibt nicht wenige Fälle, in denen die gezeigte Vorgehensweise zu falschen Rechenergebnissen führt. Ein solches Beispiel zeigt Listing 2.

Das Beispielprogramm aus Listing 2 enthält eine Schleife, deren Durchläufe voneinander abhängig sind. Es kann Situationen geben, in denen Thread #2 das 500. Array-Element berechnen will und dazu das 499. Element benötigt. Dieses wird jedoch

Listing 3

Die parallele foreach-Schleife.

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

namespace TPL_1_3
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> myList = new List<int>();
            // Daten in Liste einfügen
            for (int i = 1; i <= 10; i++)
            {
                myList.Add(i);
            }
            Console.WriteLine("Sequenziell:");
            // Normale, sequenzielle foreach-
            // Schleife
            foreach (var item in myList)
            {
                Console.Write(" {0}", item);
            }
            Console.WriteLine("\nParallel:");
            // Parallele foreach-Schleife
            Parallel.ForEach(myList, item =>
            {
                Console.Write(" {0}", item);
            });
            Console.WriteLine();
        }
    }
}
```

irgendwann von Thread #1 berechnet. Auf jeden Fall steht es noch nicht zur Verfügung, wenn Thread #2 es braucht. Leider gibt der C#-Compiler hier keine Fehlermeldung aus, sodass man nach wie vor sehr gut über den parallelen Code nachdenken muss.

Es kommt sogar noch schlimmer! Wenn Sie das Programm aus Listing 2 ausführen, werden Sie (wahrscheinlich) feststellen, dass es funktioniert. Alles ist richtig, das Ergebnis ist 1000. Was ist da los? Nun, ganz einfach. Die TPL hat festgestellt, dass in dieser Schleife (nur bis 1000) so wenig zu tun ist, dass es sich nicht lohnt, das Ganze zu parallelisieren. In diesem Fall lässt die TPL die Schleife normal sequenziell ablaufen. Das wiederum bedeutet, dass nur ein Thread im Einsatz war und dementsprechend das Ergebnis der Schleife auch richtig berechnet wurde.

Zum Testen sollte man den Wert von *nAnzahl* schrittweise auf 10 000, 100 000 und so weiter erhöhen. Irgendwann tritt der beschriebene Fehler auf. Dieses Verhalten zeigt, dass das Testen bei Programmen, die parallelen Code enthalten, nicht gerade einfacher wird. Es kann durchaus vorkommen, dass beim Testen alle Berechnungen richtig sind und erst beim Kunden fehlerhafte Ergebnisse auftreten, wenn „echte“ Daten eingesetzt werden.

Natürlich ist es auch möglich, mit der TPL eine *foreach*-Schleife zu parallelisieren. Listing 3 zeigt ein einfaches Beispiel.

Bei der Ausführung der parallelen *foreach*-Schleife aus Listing 3 ist sehr schön zu sehen, dass die Elemente nicht der Reihe nach verarbeitet werden, sondern in einer Reihenfolge, die nicht vorhersagbar ist:

```
Sequenziell:  
1 2 3 4 5 6 7 8 9 10  
Parallel:  
1 2 3 4 6 7 8 10 5 9
```

Wenn eine Schleife mit der TPL parallelisiert wird, so werden normalerweise alle Prozessorkerne benutzt, die auf dem Rechner vorhanden sind. Dies ist unabhängig davon, ob die Kerne ausgelastet sind oder sich im Wartezustand (Idle) befinden. So werden zum Beispiel auf einem Rechner mit einem QuadCore-Prozessor vier Threads und auf einem Dual Xeon QuadCore acht Threads von der TPL für parallel auszuführende Aufgaben benutzt. Diese Threads kommen übrigens aus einem speziellen Thread-Pool, sodass die Bereitstellungszeiten für die vielen Threads möglichst kurz sind. Die oben genannte Thread-Anzahl wird aber nur dann eingesetzt, wenn die Aufgabe in der Schleife groß genug ist, sodass sich eine Bereitstellung und Initialisierung der vielen Threads auch lohnt.

Listing 4

Code für den Schleifentest.

```
static void fnTest2()  
{  
    // Parallele Version  
    int n = 1000;  
    double[] arrData2 = new double[n];  
    Parallel.For(0, n, i =>  
    {  
        arrData2[i] = Math.Sqrt(i) +  
            Math.Sin((double)i / 1000.0);  
    });  
}
```

Tabelle 1

Ausführungszeiten der Schleife.

Anzahl der Durchläufe	Sequenziell	Parallel
1000	0,0517 ms	0,0732 ms
10000	0,5061 ms	0,2151 ms
100000	5,1738 ms	1,5484 ms

Trotzdem dauert es eine kurze Zeit, bis die TPL herausgefunden hat, wie viele Threads für ein bestimmtes Problem am besten verwendet werden sollten. Das bedeutet, dass eine Schleife, die mit *Parallel.For* programmiert wurde und dann aufgrund geringer Problemgröße nur in einem Thread läuft, immer etwas länger dauert als die entsprechende Schleife mit dem normalen *for*-Konstrukt aus C#. Werden jedoch mehrere Threads von der TPL eingesetzt, so kann man davon ausgehen, dass Rechenzeit eingespart wird und die Laufzeit der Schleife verkürzt ist. Lässt man den Testcode aus Listing 4 sowohl sequenziell als auch parallel laufen, erhält man für unterschiedliche Schleifentiefen die Ergebnisse aus Tabelle 1. Diese wurden auf einem QuadCore-Prozessor ermittelt.

Nicht ausgewogene Schleifen

In der alltäglichen Programmierarbeit hat man es sehr oft mit Schleifen zu tun, deren einzelne Durchläufe unterschiedlich lange dauern, siehe Listing 5.

Würde die Schleife in Listing 5 für zwei Prozessorkerne einfach in der Mitte aufgeteilt werden, dann hätte Kern 1 sehr lange etwas zu tun und Kern 2 wäre schon nach kurzer Zeit mit seiner Arbeit fertig. In diesem Fall würde die Parallelisierung so gut wie nichts bringen, da die beiden Prozes-

Listing 5

Ein unausgewogene Schleife.

```
for(int i = 0; i < 1000; i++)  
{  
    if(i < 500)  
    {  
        DoLongCalculation();  
    }  
    else  
    {  
        DoShortCalculation();  
    }  
}
```

Listing 6

Daten aggregieren.

```
double dSum = 0.0;  
double[] dA = new double[1000];  
  
// dA hier mit Daten füllen  
/ ...  
  
for(int i = 0; i < 1000; i++)  
{  
    dSum += dA[i];  
}  
  
Console.WriteLine("Summe: {0}", dSum);  
  
// ...
```

Listing 7

lock verwenden.

```
double dSum = 0.0;  
double[] dA = new double[1000];  
object objLock = new object();  
  
// dA hier mit Daten füllen  
/ ...  
  
for(int i = 0; i < 1000; i++)  
{  
    lock(objLock)  
    {  
        dSum += dA[i];  
    }  
}  
  
Console.WriteLine("Summe: {0}", dSum);  
  
// ...
```

sorkerne nicht gleichzeitig arbeiten. Aus diesem Grund wird die Schleife von der TPL automatisch in viele kleinere Arbeitseinheiten unterteilt. Immer wenn ein Thread mit einer Arbeitseinheit fertig ist, fordert er sofort die nächste an, bis die gesamte Aufgabe erledigt ist. Der Verwaltungsaufwand steigt dadurch natürlich an, aber die TPL kann beide Prozessorkerne nun etwa gleichmäßig auslasten und somit eine hohe Effizienz durch den parallelen Code erzielen.

Aggregationen

Häufig findet man in einer Software Schleifen der Ausprägung, wie sie Listing 6 zeigt. In dieser Schleife werden die Daten aus dem Array *dA* als Summe zusammengefasst. Ein Problem entsteht dann, wenn man die obige Schleife parallelisiert, denn

in diesem Fall würden mehrere Threads gleichzeitig schreibend auf die Variable *dSum* zugreifen, was natürlich nicht erlaubt ist. Hier gibt es verschiedene Lösungsmöglichkeiten. Zunächst einmal könnte man mit einem *Lock* die kritische Zeile der Addition so schützen, dass immer nur ein Thread diese Zeile ausführen kann. Bevor dieser Thread mit der Addition fertig ist, kann kein anderer Thread einen Zugriff auf die Variable *dSum* erhalten, siehe Listing 7.

Dieser Code hat jedoch den gravierenden Nachteil, dass er ziemlich langsam ist. Er ist wahrscheinlich sogar langsamer als ganz normaler sequenzieller Schleifencode.

Ein *Lock*-Statement sollte so selten wie möglich – aber so oft wie nötig – aufgerufen werden. Hier setzt die TPL an. Schleifen werden von der TPL in viele kleine Arbeitseinheiten unterteilt. Bei einer Aggregation mit der TPL wird der *ThreadLocalState* benutzt, um das Zwischenergebnis einer kleinen Arbeitseinheit in einer Lambda-Funktion aufzusummieren. Danach wird eine weitere Lambda-Funktion benötigt, um das Zwischenergebnis zur endgültigen Ergebnisvariablen zu addieren. Diese zweite Lambda-Funktion wird mit einem ganz normalen *Lock*-Element erstellt. Sie wird jedoch nur sehr selten aufgerufen, nämlich immer nur dann, wenn eine Arbeitseinheit der Schleife abgearbeitet ist. Das wiederum bedeutet, dass sich diese *Lock*-Aufrufe untereinander kaum stören werden. Ein vollständiges Beispiel einer Aggregation zeigt Listing 8.

In Listing 8 werden drei Lambda-Funktionen eingesetzt. Die erste davon initialisiert das *ThreadLocalState*-Element mit dem Wert *0.0*, also als Typ *double*. Die zweite Lambda-Funktion stellt den Schleifenrumpf dar. Hier ist zu beachten, dass die TPL drei Variablen übergibt:

- die Laufvariable der Schleife (*i*),
- ein Objekt vom Typ *ParallelLoopState* (*pls*, wird hier nicht benötigt)
- und ein Objekt vom Typ *ThreadLocalState* (*tls*).

Jeder Thread, der an der Abarbeitung der Schleife beteiligt ist, hat sein eigenes *tls*-Objekt. Somit kann man bei den Berechnungen auf die Variable *tls* aus dem Thread heraus ohne jegliches Locking zugreifen und diese zur Bildung der Zwischensumme heranziehen. Wenn nun eine Arbeitseinheit erledigt ist, wird mit der Zwischensumme (Variable *partSum*) die dritte Lambda-Funktion aufgerufen. Dort wird die Teilsumme in die endgültige Variable *dSum*

Listing 8

Daten aggregieren.

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

namespace TPL_1_5
{
    class Program
    {
        static object aggLock = new object();
        static void Main(string[] args)
        {
            // Sequenzieller Test
            double dTest = 0.0;
            for (int i = 0; i < 100000000; i++)
            {
                dTest += Math.Sqrt(i);
            }
            // Parallele Aggregation
            double dSum = 0.0;
            int iCountLock = 0;
            Parallel.For(0, 100000000,
                () => 0.0, // Initialisierung von tls
                (i, pls, tls) => // Schleifenrumpf
                {
                    tls += Math.Sqrt(i);
                    return tls;
                },
                (partSum) => // Verarbeitung der Zwischensumme
                {
                    lock (aggLock)
                    {
                        dSum += partSum;
                        iCountLock++;
                    }
                });
            Console.WriteLine("Test: {0} Parallel: {1}", dTest, dSum);
            Console.WriteLine("Anzahl der Locks: {0}", iCountLock);
        }
    }
}
```

addiert. Hier muss ganz normales Locking benutzt werden, um den Zugriff auf die Variable *dSum* zu synchronisieren.

Im Beispielprogramm wird in der dritten Lambda-Funktion auch noch die Variable *iCountLock* um 1 erhöht. Diese Variable hat für das Beispiel keine Funktionalität. Sie gibt zum Schluss jedoch an, wie oft die dritte Lambda-Funktion aufgerufen wurde und somit das Locking ausgeführt werden musste.

Bei meinen Experimenten habe ich festgestellt, dass die abschließende Lambda-Funktion im Beispiel aus Listing 8 etwa 25-

bis 35-mal aufgerufen wurde. Somit kann der Lock als eher unkritisch angesehen werden.

Wenn Sie das Beispielprogramm ausführen, fällt Ihnen sofort noch eine unangenehme Kleinigkeit auf. Das sequenzielle Testergebnis unterscheidet sich in diesem Fall vom parallelen Ergebnis! Hierbei handelt es sich allerdings nicht um einen Programmierfehler, sondern um die normale numerische Ungenauigkeit bei der Addition von Fließkommazahlen.

Wie vorhin schon erwähnt, werden möglichst alle Kerne von der TPL herangezogen.

Listing 9

Die Anzahl der Threads begrenzen.

```
// Nur 2 Prozessorkerne benutzen
var options = new ParallelOptions { MaxDegreeOfParallelism = 2 };
Parallel.For(0, 10000, options, i =>
{
    dRes[i] = Math.Sqrt(i);
});
```

Listing 10

Codebereiche parallel ausführen.

```
// ...

class Program
{
    private static double[] dY1 = new double[10000];
    private static double[] dY2 = new double[10000];
    private static double[] dX = new double[10000];

    static void Main(string[] args)
    {
        // ...
        bool bOK = CalcData();
        // ...
    }

    static bool CalcData()
    {
        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++)
                {
                    dY2[i] = Math.Sqrt(dX[i] / 2.0) + Math.Cos(dX[i]);
                }
            });
        return true;
    }
}
```

gen, um eine Schleife parallel ablaufen zu lassen. In manchen Fällen kann es jedoch sinnvoll sein, die Thread-Anzahl zu erhöhen oder zu erniedrigen.

Dies ermöglicht die Klasse *ParallelOptions* über die Klasseeigenschaft *MaxDegreeOfParallelism*. Mit dem Wert *-1* wird die maximale Anzahl der Kerne für die Berechnung ausgenutzt.

Ein entsprechend initialisiertes Objekt der Klasse *ParallelOptions* wird dann in der

Parallel.For-Methode verwendet, wie in Listing 9 zu sehen.

Parallele Codebereiche

Wenn keine Schleifenkonstrukte vorliegen, ist es oft möglich, einzelne Codebereiche parallel ablaufen zu lassen.

Natürlich müssen diese Codebereiche unabhängig voneinander sein. Die hier einzusetzende Methode *Invoke* ist in der *Parallel*-Klasse der TPL zu finden. *Parallel.Invoke*

ist in der Lage, bis zu zehn Codebereiche parallel auszuführen. Allerdings wird auch hier wieder, wie bei den Schleifen, synchronisiert. Das bedeutet, es geht nach dem *Parallel.Invoke*-Aufruf erst dann weiter, wenn alle Codebereiche vollständig ausgeführt sind. Listing 10 zeigt einen Teil des Beispiels für die parallele Verarbeitung von Codebereichen.

In Listing 10 laufen zwei Codebereiche parallel ab, die jeweils in einer eigenen Lambda-Funktion angegeben wurden. Beide Lambdas haben den vollen Zugriff auf die Daten-Arrays. Dies ist in diesem Fall jedoch kein Problem, da jeder Thread ein anderes Array schreibt und das Array *dX* von allen Threads nur gelesen wird.

Zusammenfassung

Im .NET Framework 4 gibt es die neue Bibliothek TPL, welche die Erstellung von parallel ausgeführtem Code deutlich erleichtert. Der aufwendige Zusatzcode bei der Programmierung von Anwendungen mit Multithreading wird ersetzt durch leicht zu lesenden Code, der genau dort die Parallelanweisungen enthält, wo sie auch zur Ausführung kommen.

Trotzdem muss der Entwickler weiterhin über jede einzelne Zeile Parallelcode sehr genau nachdenken. Und natürlich ist eins auch in einer parallelen Codewelt angesagt: Testen, testen, testen.

Der zweite Teil dieser Serie geht auf die Klasse *Task* der TPL ein, da die taskorientierte Programmierung das komplexe Asynchron-Pattern auf einfache Weise ersetzen kann. [ml]

- [1] Bernd Marquardt, Mach's doch gleichzeitig, Parallelprogrammierung mit OpenMP und C++/CLI, dotnetpro 4/2009, S. 96ff., www.dotnetpro.de/A0904OpenMP
- [2] Klaus Aschenbrenner, Das Message Passing Interface, Windows HPC Server 2008: Kommunikation via MPI, dotnetpro 2/2009, S. 85ff., www.dotnetpro.de/A0902MPI
- [3] Klaus Aschenbrenner, MPI in der Praxis, Windows HPC Server 2008: Kommunikation via MPI, dotnetpro 3/2009, S. 68ff., www.dotnetpro.de/A0903MPI
- [4] Bernd Marquardt, MSDN-Webcasts, Parallelprogrammierung mit der Parallel Pattern Library (PPL) (Teil 1 und Teil 2), www.dotnetpro.de/SL1009TPL1 und www.dotnetpro.de/SL1009TPL2
- [5] Ralf Westphal, Auf dem Weg zu einer neuen Architektur, Grundlagen skalierbarer Parallelverarbeitung, dotnetpro 2/2009, Seite 128ff., www.dotnetpro.de/A0902ArchitekturKolumne