Workshop: Task Parallel Library (TPL)

# Einführung

Bei der TPL handelt es sich um eine Bibliothek für die Entwicklung multi-thread-fähiger Anwendungen. Sie steht seit Visual Studio 2010 und dem .NET Framework 4.0 zur Verfügung. Der große Vorteil für den Entwickler liegt darin, dass auf das Management jedes einzelnen Threads verzichtet werden kann. Stattdessen liegt der Fokus auf der Aufgabenstellung und der Definition von Tasks und deren Synchronisation. Durch die TPL erfolgt das Erstellen der Threads automatisch während der Laufzeit unter Berücksichtigung der optimalen Hardwareausnutzung. D.h. bei einem Vierkern-Prozessor würde der Task automatisch in vier Threads aufgeteilt und abgearbeitet, bei einem Achtkern-Prozessor dementsprechend automatisch in acht Threads usw. [1].

Grundsätzlich kann zwischen Datenparallelismus und Taskparallelismus unterschieden werden. Im folgenden Workshop wird auf diese beiden Konzepte in Zusammenhang mit der TPL näher eingegangen und jeweils mit einem bzw. mehreren Beispielen veranschaulicht.

# Data Parallelism

Unter Datenparallelismus versteht man eine Anwendung, in denen der gleiche Vorgang gleichzeitig d.h. parallel auf einen Teilbereich der gesamten Datenmenge (z.B. Array) angewendet wird. Dadurch können mehrere Threads gleichzeitig auf verschiedene Datensegmente angewendet werden [2].

In der TPL wird dieser Anwendungsfall durch die Klasse *System.Threading.Tasks.Parallel* unterstützt. Diese Klasse stellt methodenbasierte parallele Implementierungen von Schleifen bereit. Durch Verwendung der TPL müssen keine Threads oder Arbeitsaufgaben in die Warteschlange eingereiht werden, die Schleifenlogik für eine parallele *for* oder *foreach*-Schleife bleibt weitgehend die gleiche wie bei einer sequenziellen Implementierung. Grundsätzlich sind auch keine Sperren erforderlich.

Um die Einfachheit der TPL zu veranschaulichen ist in *Abbildung 1* ein einfaches Beispiel einer sequenziellen *foreach*-Schleife und deren paralleler Implementierung abgebildet. Bei der parallelen-Schleife wird die Datenquelle (z.B. Array) partitioniert, sodass mehrere Teile der Datenquelle gleichzeitig bearbeitet werden können. Der Taskplaner arbeitet dabei im Hintergrund und partitioniert die Aufgabe mit Berücksichtigung der Systemressourcen und der Arbeitsauslastung [2]

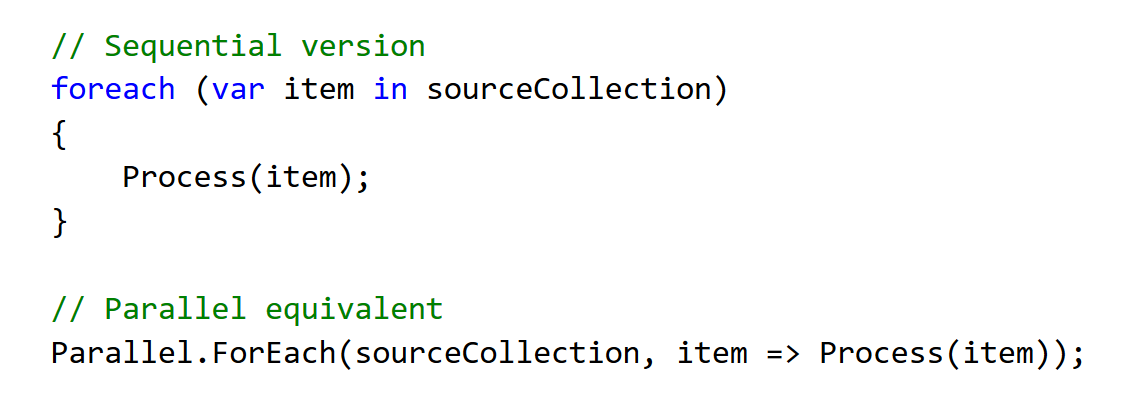


Abbildung 1 - Sequenzielle und parallele Implementierung (mit TPL) einer einfachen foreach-Schleife [2]

**Weitere Informationen:**

Die parallelen Implementierungen der *for* bzw. *foreach*-Methode verfügen zudem über Überlagerungen für:

* das Anhalten bzw. Unterbrechen der Schleifenausführung,
* die Zustandsüberwachung der Schleife in anderen Threads,
* das Beibehalten des lokalen Threadzustandes,
* das Abschließen lokaler Threadobjekte,
* die Steuerung des Parallelitätsgrades [2].

Es folgen zwei Beispiele für die parallele Implementierung von Schleifen. Das erste Beispiel behandelt das Thema threadlokale Variablen, beim zweiten Beispiel wird Exception Handling beschrieben.Weitere Beispiele für die Implementierung von Datenparallelismus mit Hilfe der TPL sind unter folgendem Link abrufbar: <https://msdn.microsoft.com/de-DE/library/dd537608(v=vs.110).aspx> [aufgerufen am 18.11.2017]

##### Parallele for-Schleife mit threadlokaler Variablen

Threadlokale Variablen werden verwendet, um den Status in jeder Aufgabe speichern und abrufen zu können. Der Vorteil liegt darin, dass dadurch vermieden werden kann in jedem Schleifendurchlauf auf eine freigegebene Ressource zu schreiben. Stattdessen wird der Wert berechnet und gespeichert bis alle Iterationen für die Aufgabe abgeschlossen sind. Das Endergebnis kann danach einmal an die freigegebene Ressource geschrieben werden bzw. an eine andere Methode übergeben werden. Im nachfolgenden Absatz ist ein Beispielcode [2], welcher die Summe der Werte in einem Array mit einer Million Elemente berechnet.

Die ersten beiden Übergabewerte der parallelen for-Schleife stellen den Anfangs- bzw. Enditerationswert dar. In dieser Überlagerung ist der dritte Parameter jener, mit der der lokale Zustand initialisiert wird. D.h. eine Variable mit einer Lebensdauer vom Anfang bis zum Ende der Iteration im aktuellen Thread. Der dritte Parameter im unten angeführten Beispiel ist vom Typ *Func<TResult>,* welcher den threadlokalen Zustand speichert. Der Ausdruck *() => 0* initialisiert die lokale Threadvariable auf Null. Der vierte Parameter definiert die Schleifenlogik mittels Lambdaausdruck [2].

class Test

{

static void Main()

{

int[] nums = Enumerable.Range(0, 1000000).ToArray();

long total = 0;

// Use type parameter to make subtotal a long, not an int

Parallel.For<long>(0, nums.Length, () => 0, (j, loop, subtotal) =>

{

subtotal += nums[j];

return subtotal;

},

(x) => Interlocked.Add(ref total, x)

);

Console.WriteLine("The total is {0:N0}", total);

Console.WriteLine("Press any key to exit");

Console.ReadKey();

}

}

##### Exception Handling in parallelen Schleifen

Die parallelen *for*- bzw. *foreach<TSource>-*Überladungen verfügen über keinen speziellen Mechanismus zum Exception Handling. Eine nicht behandelte Exception führt zum Abbruch der Schleife. Eine Ausnahmebehandlung in parallelen Schleifen behandelt den Fall, in dem ähnliche Ausnahmen in mehreren Threads gleichzeitig ausgelöst werden können, sowie den Fall, dass eine Ausnahme die in einem Thread ausgelöst wird eine Ausnahme in einem anderen Thread zur Folge hat. Beide Fälle können behandelt werden in dem alle Ausnahmen innerhalb der Schleife mit einer *System.AggregateException* umschlossen werden [2].

Der folgende Codeteil [2] zeigt wie eine oder mehrere Ausnahmen mit einer *System.AggregateException* umschlossen werden können. Der Aufrufer kann entscheiden welche Exceptions behandelt werden.

try

{

ProcessDataInParallel(data);

}

catch (AggregateException ae)

{

// This is where you can choose which exceptions to handle.

foreach (var ex in ae.InnerExceptions)

{

if (ex is ArgumentException)

Console.WriteLine(ex.Message);

else

throw ex;

}

In der Funktion selbst bzw. der Schleifenlogik kann die Implementierung folgendermaßen aussehen:

private static void ProcessDataInParallel(byte[] data)

{

// Use ConcurrentQueue to enable safe enqueueing from multiple threads.

var exceptions = new ConcurrentQueue<Exception>();

// Execute the complete loop and capture all exceptions.

Parallel.ForEach(data, d =>

{

try

{

// Cause a few exceptions, but not too many.

if (d < 0x3)

throw new ArgumentException(String.Format("value is {0:x}. Elements must be greater than 0x3.", d));

else

Console.Write(d + " ");

}

// Store the exception and continue with the loop.

catch (Exception e) { exceptions.Enqueue(e); }

});

// Throw the exceptions here after the loop completes.

if (exceptions.Count > 0) throw new AggregateException(exceptions);

}

# Task Parallelism

Beim Task/Aufgaben Parallelismus werden Tasks (Einheiten von Arbeit) Parallel ausgeführt und abgearbeitet. Dies können zum Beispiel Tasks sein, die mit den gleichen Daten unterschiedliche Aufgaben/Berechnungen durchführen (Wenn die Ergebnisse unabhängig sind und nicht das gleiche Ziel als Speicherort dient). Es können aber auch Tasks sein die völlig unabhängig sind und im Hintergrund berechnet werden sollen. Die TPL und der Task Parallelismus vereinfacht dabei die Anwendung und Verwaltung von Threads. Die Library bietet dabei zwei wesentliche Vorteile:

* Effiziente und skalierbare Verwendung von Systemressourcen [2]:

Da die Anzahl der Threads vom Framework bestimmt wird und die Auslastung auf das System angepasst wird.

* Stärker programmgesteuerte Kontrolle als bei Threads oder Arbeitsaufgaben:

Das Framework stellt einen Umfangreichen Satz von APIs zur Verfügung. Mit diesem ist das Handeln von Tasks vereinfacht.

Es können beliebig viele Tasks erstellt werden. Diese werden im *ThreadPool* angelegt und vom Framework gemanagte. Der *ThreadPool* verwaltete dabei die Anzahl der Threads und versucht den Durchsatz zu maximieren. Dabei wird vom Framework darauf geachtet, dass die CPU Kerne bestmöglich ausgenutzt werden.

Daher ist es auch nicht notwendig explizit Threads zu erstellen und zu verwalten [2]. In den folgenden Seiten werden die wichtigsten Funktionen des Task Parallelismus kurz beschrieben.

##### Impliziertes Erstellen und Ausführen von Aufgaben/Tasks

Dies wird mit der Methode *Parallel.Invoke()* ausgeführt. Diese Methode bietet die Möglichkeit eine beliebige Anzahl von Tasks gleichzeitig auszuführen. Für jede Aufgabe muss dazu ein Action-Delegate übergeben werden. Ein Action-Delegate ist die Kapselung einer Methode, die über keinen Parameter verfügt und keinen Wert zurückgibt. Sie wird verwendet, um Methoden als Parameter zu übergeben. Am einfachsten werden Action-Delegates mit Lambda-Ausdrücken erstellt.

Im Lambda Ausdruck kann entweder eine benannte Funktion oder Inline-Code übergeben werden [2].

Hier ein einfaches Beispiel einer implizierten Ausführung von zwei parallelen Tasks

Parallel.Invoke(

() => DoSomeWork(), //erste Methode zur parallelen Ausführung

() => {Console.WriteLine("Test");} //inline Code zum parallelen Ausführen

);

Der *Parallel.Invoke()*-Methode können noch *ParallelOptions* als Parameter mitgegeben werden. Diese können Eigenschaften wie einen *CancellationTokan* (zum Abbrechen eines Tasks siehe später), die maximale Anzahl an Parallelisierung der Tasks oder ein *TaskScheduler* beinhalten. Hier ein kurzes Beispiel [2].



CancellationTokenSource cts = new CancellationTokenSource();

CancellationToken ct = cts.Token; //erstellen eines Abbruch Tokens zum Abbrechen eines Tasks

//erstellen eines ParallelOptions mit Einstellungen für den Task

ParallelOptions po = new ParallelOptions {

CancellationToken = ct,

MaxDegreeOfParallelism = System.Environment.ProcessorCount //max. Tasks

};

//hier werden die Options an den Task weitergegeben

Parallel.Invoke(po,() => doWork(„Hallo“));

##### Explizites Erstellen und ausführen von Aufgaben/Tasks

Beim Expliziten erstellen und starten von Tasks können Tasks erstellt werden und zu einem späteren Zeitpunkt gestartet werden. Bei expliziter Erstellung von Tasks wird ein Task Objekt erzeugt. Mit diesen kann jederzeit auf zum Beispiel den Status des Tasks zugegriffen werden.

Den Task Objekten werden Delegates zugewiesen. Die Delegates enthält enthalten den vom Task auszuführenden Code. Diese Delegatse können als benannter Delegat, als anonyme Methode oder als Lambda-Ausdruck angegeben werden.

Bei den Task Objekten unterscheidet man zwischen Tasks mit und Tasks ohne Rückgabewerten ([*Task*](https://msdn.microsoft.com/de-de/library/system.threading.tasks.task(v=vs.110).aspx) und [*Task<TResult>*](https://msdn.microsoft.com/de-de/library/dd321424(v=vs.110).aspx)) [2].

###### Task starten

Das Starten eines Tasks kann auf verschiedene Arten erfolgen. Folgend werden drei Methoden beschrieben.

Zum Starten eines Tasks kann zuerst ein neuer Task mit dem Delegate erstellt werden. Danach kann mit der *Task.Start()-*Methode der Task gestartet werden [2].

Hier sehen sie ein Beispiel eines Tasks ohne Rückgabewert

//erstellen eines neuen Tasks mit dem Delegate

Task task = new Task( () => Console.WriteLine("Hello from task."));

task.Start(); //starten des Tasks

Weiters kann dem Task direkt beim Aufruf in die *Task.Run*-Methode der Delegate übergeben werden. Dies startet den Task direkt nach dem Aufruf [2].

Task task = Task.Run( () => Console.WriteLine("Hello from task."));

Eine weitere und zu bevorzugende Methode, um einen Task direckt zu starten ist die *Task.Factory.StartNew()*-Methode. Wie im folgenden Beispiel dargestellt [2].

Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello from task."));

###### Task mit Rückgabewert

Soll ein Task einen Rückgabe Wert liefern, kann dies mit der Klasse [*Task<TResult>*](https://msdn.microsoft.com/de-de/library/dd321468(v=vs.110).aspx) geschehen. Dabei wird dem Task die Klasse des Resultats übergeben. Der Task kann dabei wieder wie vorher beschrieben gestartet werden. Durch die Eigenschaft *Task.Result* kann das Ergebnis des Tasks abgefragt werden. Wird vor Beendigung des Tasks auf die *Task.Result*-Methode zugegriffen wird der Thread der das Resultat abfragt nicht fortgesetzt bis das Ergebnis vorliegt [2].

Im folgenden ein kurzes Beispiel zu Tasks mit Rückgabewert:

//erstellen und ausführen eines Tasks welcher ein neues Objekt der Klasse Test zurück gibt.

Task<Test> task = Task<Test>.Factory.StartNew(() =>

{

string s = "TPL Workshop";

double d = 1.0;

return new Test { Name = s, Number = d };

});

Test test = task.Result; //hier wird der return-value abgefragt

//Klasse als Rückgabewert

class Test

{

public string Name { get; set; }

public double Number { get; set; }

}

###### Warten Auf Tasks

Auf Tasks kann gewartet werden, bis diese abgeschlossen sind. Dies kann durch den Aufruf der Methode *Task.Wait()* geschen. Weiters kann auf ein Array mit Tasks gewarted werden. Dies erfolgt mit der Methode *Task.WaitAll()* [2].

Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello from task."));

task.Wait() //warten bis der Task abgearbeitet ist.

###### Task Array

Tasks können auch in Arrays verwaltet werden.

Im folgenden Code Beispiel werden in einem Array Tasks gespeichert und gestartet. Danach wird mit *Task.WaitAll()* gewartet bis alle Tasks fertig sind [2].

Task[] taskArray = new Task[10]; //erstellen eines leeren Arrays

for (int i = 0; i < taskArray.Length; i++) {

//über das gesamte array iterieren und die Tasks starten

taskArray[i] = Task.Factory.StartNew((Object obj) =>

{

//code des Tasks

CustomData data = obj as CustomData; //parse

if (data == null) return; //check if not null

data.ThreadNum = Thread.CurrentThread.ManagedThreadId;

//ausgeben einiger Werte und der Thread ID

Console.WriteLine("Task #{0} created at {1} on thread #{2}.",

data.Name, data.CreationTime, data.ThreadNum);

},

//Übergabe Objekt an die Lambda Funktion

new CustomData() {Name = i, CreationTime = DateTime.Now.Ticks} );

}

Task.WaitAll(taskArray); //warten bis alle Tasks feting sind

###### Tasks abbrechen

Das Abbrechen von Tasks wird mithilfe von Abbruch-Token realisiert. Um einen Task abzubrechen muss die Methode [*CancellationTokenSource.Cancel*](https://msdn.microsoft.com/de-de/library/dd321955(v=vs.110).aspx)() aufgerufen werden.

Dabei gibt es zwei Optionen den Task zu beenden [2]:

* Der Task wird beendet (z.B. mit return) und kehrt zurück. Dabei wird der Zustand des Tasks auf *TaskStatus.RunToCompletion* gesetzt und nicht auf *TaskStatus.Canceled*.

Es wird eine *OperationCanceldException* ausgelöst. Dabei wird auch der *TaskStatus* auf *Canceled* gesetzt. Das erzeugen dieser Exception kann durch den Methodenaufruf *CancellationToken.*[*ThrowIfCancellationRequested*](https://msdn.microsoft.com/de-de/library/system.threading.cancellationtoken.throwifcancellationrequested(v=vs.110).aspx)*()* erzeugt werden.

Der Cancel Token muss wie im folgenden Beispiel erstellte werden und kann dann wie in dem Task dargestellt behandelt werden [2]:

var tokenSource = new CancellationTokenSource();

CancellationToken ct = tokenSource.Token; //erstellen des CancellationToken

var task = Task.Factory.StartNew(() =>

{

// abfrage ob der Task bereits abgebrochen wurde

ct.ThrowIfCancellationRequested();

bool moreToDo = true;

while (moreToDo)

{

// Abfrage ob der Task gecanceld wurde um auch ein mögliches aufräumen vor dem beenden durchzuführen

if (ct.IsCancellationRequested)

{

//mögliches clean up behandeln

// Methode zum werfen der Exception aufrufen

ct.ThrowIfCancellationRequested();

}

}

}, tokenSource2.Token); //Übergeben des Token an den Task

//aufruf der Cancel Methode zum stoppen des Tasks

tokenSource.Cancel();

###### Continuing Tasking

Beim Continuing Tasking wird sobald ein Task beendet ist, ein weiterer Task ausgeführt, mit den Ergebnissen des vorherigen Tasks. Dies kann durch den Aufruf der Methode *Task.ContinueWith()* erreicht werden. Der Übergabewert des vorherigen Tasks ist dabei der Task selber, es kann daher mit *Task.Result* auf das Ergebnis des vorherigen Tasks zugegriffen werden [2].

Im folgenden Beispiel wird zuerst ein Task gestartet, der ein Array mit Zahlen erstellt. Dieses Array wird an den nächsten Task weitergegeben und dort weiterverarbeitet. Dem dritten Task wird ein Tuple übergeben und dieser gibt die Daten in der Konsole aus [2].

//starten des ersten Tasks, welcher zufallszahlen erzeugt

var getData = Task.Factory.StartNew(() => {

Random rnd = new Random();

int[] values = new int[100];

for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)

{

values[ctr] = rnd.Next();

}

return values; //der return value ist ein Array

} );

//nach fertigstellen des ersten Tasks wird mit dem Zweiten begonnen

var processData = getData.ContinueWith((x) => {

//über Task.Result kann auf das Ergebnis des vorherigen Tasks zugegriffen werden

int n = x.Result.Length;

long sum = 0;

double mean;

for (int ctr = 0; ctr <= x.Result.GetUpperBound(0); ctr++)

{

sum += x.Result[ctr];

}

mean = sum / (double) n;

return Tuple.Create(n, sum, mean);

//als Ergebnis wir dein Tuple mit dem Mittelwert,…

zurückgegeben

} );

//nach fertigstellen des zweiten Tasks wird mit dem Dritten begonnen

var displayData = processData.ContinueWith((x) => {

//dieser gibt einen String zurück

return String.Format("N={0:N0}, Total = {1:N0}, Mean = {2:N2}", x.Result.Item1, x.Result.Item2, x.Result.Item3);

} );

//zum Abschluss wird das Resultat des letzten Tasks ausgegeben

Console.WriteLine(displayData.Result);

###### Ausnahmebehandlung (Exception Handling) in Tasks

Alle Exceptions die nicht in dem Task behandelt werden, werden an den aufrufenden Thread weitergegeben. Exceptions können dabei von der *Task.Wait()-*Methode mit *try/catch* abgefangen werden. Weiters kann auf die Exception eines Tasks mit der Property *Task.Exception* zugegriffen werden.

Wird eine Exception erzeugt, die nicht im Thread abgefangen wird, wird eine *AggregateException* ausgelöst. Diese beinhaltet sogenannte *InnerExceptions*. In diesen werden die vom Task geworfenen Exceptions gespeichert [2].

Hier ein Beispiel mit *Task.Wait()* und dem Abfangen einer Exception aus einem Task

var task = Task.Run( () => { throw new CustomException("Exception expected");});

try

{

task.Wait(); //warten bis der Thread fertig ist im try block

}

catch (AggregateException ae) //abfangen der AggregateException

{

foreach (var e in ae.InnerExceptions) {

// hier wird über alle InnerExceptions Iteriert. (Falls mehrere existieren)

if (e is CustomException) {

Console.WriteLine(e.Message); //ausgabe der Exception

}

// falls es eine andere Exception ist wird diese weitergegeben

else {

throw;

}

}

}

Weiters ein Beispiel zum Abfrangen von Exceptions mithilfe der *Task.Exception* Property des Tasks.

var task = Task.Run( () => { throw new CustomException("Exception expected");});

while(! task.IsCompleted) {} //warten bis der Task fertig ist

if (task.Status == TaskStatus.Faulted) {

foreach (var e in task1.Exception.InnerExceptions) {

// Handle the custom exception.

if (e is CustomException) {

Console.WriteLine(e.Message);

}

// Rethrow any other exception.

else {

throw e;

}

}

}

# Datenfluss (TPL Dataflow)

Mit dieser Library ist es möglich einfache Datenfluss und Pipelineaufgaben zu lösen. Man verwendet die Library wenn mehrere Vorgänge asynchron miteinander kommunizieren sollen oder wenn aktuell verfügbare Daten verarbeitet werden müssen (auch mehrere Datensätze parallel).

Die Datenfluss Library eine sehr wenig genutzte Library (nur 379 Frage auf Stack Overflow stand 16.11.2017) und daher hier nur kurz beschrieben. Die Dataflow Library muss in Visual Studio durch NuGet nachinstalliert werden (*Microsoft.Tpl.Dataflow*) und wurde erst mit .NET 4.5 eingeführt [2].

Das Verarbeiten von Daten wird dabei in Unterschiedliche Blöcke (Blocks) unterteilt. Das Framework bietet dabei drei Gruppen von Blocken. Die Buffer Blocks, Execution Blocks und Grouping Blocks. Ein Block besteht Grundsätzlich aus zwei Elementen einer Buffer und einer Task Komponente. Die Unterschiedlichen Blocke unterscheiden sich dabei in der Anzahl der Buffer und Task Komponenten. Die Buffer dienen zum Buffern von eingehenden Daten oder von Ausgehenden Daten. Ein JoinBlock besitzt zum Beispiel mehrere Buffer. Diese Blöcke beliebig verbunden und können dadurch Daten parallel verarbeiten [3].

Hier ein Beispiel das einen Anwendungsfall der Dataflow Library veranschaulichen soll ().

Am Beginn steht eine Datenquelle. Dies kann zum Beispiel das Laden einer Datei aus den Internet sein oder eine Message die über eine Queue ankommt (z.B. via RabbitMQ). Die Daten werden im ersten Block (Transform) transformiert. Dabei können die Daten von einer XML oder JSON Dateien in ein Objekt umgewandelt werden.

Dach werden die Daten an die nächsten Blöcke (Process) übergeben, die etwa die Daten parallel weiterverarbeiten, Berechnungen durchführen oder ähnliches. Danach können die Ergebnisse wieder in einem Block (Collect) gesammelt werden. Die gesammelten Daten werden danach wieder an einen Transform Bock weitergeben und in ein Format geparst in dem es wieder ausgegeben werden kann [4].

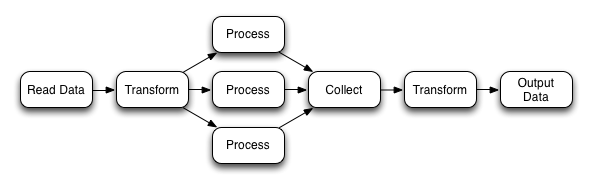


Abbildung 2: Möglicher Ablauf eines Datenflusses zur Veranschaulichung [4]

Hier noch ein kurzes Beispiel wie ein solcher Block verwendet werden kann. Zuerst wird ein Block mit einem Task erzeugt. Danach werden in einer Schleife Daten an den Block weitergegeben. Diese Daten werden vom Block verarbeitet [2].

//hier wird ein neuer Block erzeugt der eine Variable auf die Konsole schreibt

var actionBlock = new ActionBlock<int>(n => Console.WriteLine(n));

.

for (int i = 0; i < 3; i++)

{

//hier werden mehrere male Werte an den Block übergeben.

actionBlock.Post(i \* 10);

}

Für mehr Informationen zur Dataflow Library siehe in den Weiterführenden Links.

# Mögliche Pitfalls bei der Verwendung von TPL

Im folgenden Kapitel werden einige Stolperfallen kurz beschrieben. Diese Pitfalls sind genauer in der Microsoft Dokumentation nachzulesen und sollen nur einen kurzen Überblick geben [2]:

* Gehen Sie nicht davon aus, dass eine parallele Ausführung immer schneller ist.

Dies kann vorkommen, wenn zum Beispiel eine Schleife wenig Iteratoren beinhaltet und diese schnell abgearbeitet sind. Da das Erstellen und verwalten der Threads auch Zeit benötigt kann das Verwalten der Threads mehr Zeit in Anspruch nehmen als die Parallelisierung Vorteile bringt.

* Vermeiden Sie es, in gemeinsam genutzte Speicherpositionen zu schreiben.

Bei Zugriff mehrerer Threads auf statische oder Klassen Variablen kann dies zu Racebedingungen führen. Dies kann mit Sperren und Synchronisation vermieden werden, bedeutet jedoch auch eine schlechtere Leistung. Um dies zu vermeiden sollten die Schleifen überladen werden oder *System.Threading.ThreadLocal<T>-*Variablen verwendet werden.

* Vermeiden Sie eine zu starke Parallelisierung.

Bei der Parallelisierung handelt man sich Mehrkosten für Partitionierung oder Synchronisation ein. Weiters ist die Parallelisierung auch durch die Anzahl an Keren am PC eingeschränkt. Deshalb kann zu starke Parallelisierung auch einen Nachteil bringen.

* Vermeiden Sie den Aufruf nicht threadsicherer Methoden.

Die Ausführung von nicht Thread Sicheren Methoden kann zu Datenbeschädigungen führen: Wenn zum Beispiel mehrere Threads mit *WriteByte* auf einen *FileStream* zugreifen.

* Beschränken Sie Aufrufe auf threadsichere Methoden.

Threadsichere Methoden können von mehreren Threads gleichzeitig Aufgerufen werden. Dies kann jedoch auch zur Verlangsamung führen, zum Beispiel bei der damit verbundenen Synchronisierung.

* Beachten Sie Threadaffinitätsprobleme

Gewisser Code muss von einem bestimmten Thread ausgeführt werden. Als Beispiel können bei WPF oder Windows Forms Anwendungen auf Steuerelemente nur der Ersteller Thread zugreifen. Es kann also keine Aktualisierung von Steuerelementen aus parallelen Schleifen stattfinden (außer der Threadplaner wird anders konfiguriert).

* Gehen Sie nicht davon aus, dass Iterationen von "ForEach", "For" und "ForAll" immer parallel ausgeführt werden.

Parallele Schleifen werden nicht immer zwingend parallel ausgeführt. Es sollten also keine Schleifen programmiert werden, in denen die Korrektheit von der parallelen Ausführung abhängig ist.

* Vermeiden der Ausführung paralleler Schleifen im UI-Thread

Werden parallele Schleifen im UI Thread ausgeführt und werden dabei Aktualisierungen der Benutzeroberfläche durchgeführt kann dies zu Zustandsbeschädigung, verzögerten Updates und sogar Deadlocks führen.

# Weiterführende Literatur

* Offizielle C# Dokumentation:

<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl> [aufgerufen am 19.11.2017]

* Visual C# - Das umfassende Handbuch (Kapitel 11.3 TPL)

<http://openbook.rheinwerk-verlag.de/visual_csharp_2010/visual_csharp_2010_11_003.htm>

[aufgerufen am 19.11.2017]

* Einführung in Dataflow

<http://www.michaelfcollins3.me/blog/2013/07/18/introduction-to-the-tpl-dataflow-framework.html>

[aufgerufen am 19.11.2017]

# Literaturverzeichnis

|  |  |
| --- | --- |
| [1] | A. Kühnel, Visual C# 2010, Bonn: Galileo Computing, 2010. |
| [2] | Microsoft, „https://docs.microsoft.com,“ Microsoft, 30 3 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl. [Zugriff am 19 11 2017]. |
| [3] | S. Cleary, „stephencleary,“ 20 09 2012. [Online]. Available: https://blog.stephencleary.com/2012/09/introduction-to-dataflow-part-1.html. [Zugriff am 19 11 2017]. |
| [4] | M. F. Collins, „Introduction to the TPL Dataflow Framework,“ 18 07 2013. [Online]. Available: http://www.michaelfcollins3.me/blog/2013/07/18/introduction-to-the-tpl-dataflow-framework.html. [Zugriff am 19 11 2017]. |