Ewald Hofman

Parallel Computing voor .NET (PLINQ)

WANNEER WEL EN WANNEER NIET INZETTEN

Computerfabrikanten leveren extra processorsnelheid door het plaatsen van meerdere processors. De meeste applicaties zijn niet ontworpen om van deze extra processors gebruik te maken. Microsoft komt met de Parallel FX Library, die het mogelijk maakt hier op een eenvoudige manier gebruik van te maken zonder al te veel aanpassingen aan jouw huidige code. Dit artikel beschrijft hoe deze techniek werkt en hoe je hier als ontwikkelaar mee kunt werken.

Als je tegenwoordig een nieuwe computer aanschaft, is deze vaak uitgerust met twee of vier processors, de zogenaamde dual of quad-core computers. In het verleden kon de chipfabrikant een processor sneller maken door onder andere meer transistoren in de component te plaatsen. De transistoren en de schakelingen tussen de transistoren zijn tegenwoordig zo klein geworden dat fabrikanten tegen fysische beperkingen zijn opgelopen. Het is daarom niet zo eenvoudig meer om de processors te versnellen. Als oplossing voor dit probleem hebben fabrikanten gekozen voor het plaatsen van meerdere processors in één computer. Momenteel is de standaard de dual-core, maar in de nabije toekomst zal een standaardcomputer vier, acht of zelfs meer processors bevatten. Het plaatsen van meerdere processors betekent niet automatisch dat een applicatie sneller is. Doordat de meeste algoritmes in applicaties die we als ontwikkelaar schrijven sequentieel zijn opgezet, voert de computer dit algoritme in slechts één van de processors uit. De andere processors zullen niet worden belast. Om een algoritme gebruik te laten maken van alle processors, moet de applicatie multi-threaded, of parallel, worden geprogrammeerd. Het operating system zal dan de threads over de verschillende processors verdelen. Het bouwen van een parallel opgezet algoritme is echter lastig. Je moet ineens rekening houden met allerlei randvoorwaarden om de werking correct te maken, zoals races en locking. Met het gebruik van de Parallel FX Library, een uitbreiding op het .NET Framework 3.5, kun je met een paar eenvoudige aanpassingen loops en LINQ parallel maken.

```
public static void CodeVoorbeeld1()
{
   DateTime start = DateTime.Now;

   // Doorloop een processor intensieve instructie 200 keer
   for (int i = 0; i < 200; i++)
   {
      ProcessorIntensieveInstructie(i);
   };

   Console.WriteLine("Niet-parallel: {0:N0} ms",
      DateTime.Now.Subtract(start).TotalMilliseconds);
}

private static void ProcessorIntensieveInstructie(int index)
{
   for (int i = 0; i < 1000000; i++)
   {
      for (int j = 0; j < 10; j++) { int a = i + j; }
   }
}
```

Codevoorbeeld 1

Loops

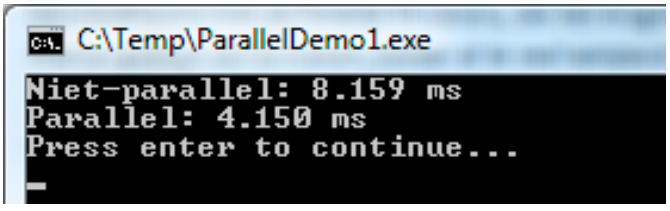
Je kunt zowel een for-loop, een foreach-loop, als een do-while-loop parallel maken met de parallelle extension. Dit artikel beschrijft het parallel laten uitvoeren van een for-loop. De overige loop- constructies zijn analoog aan de for-loop parallel te maken. Codevoorbeeld 1 toont een zeer eenvoudige applicatie die in een for-loop een processorintensieve actie doet.

Om codevoorbeeld 1 parallel te laten verlopen, moet je de for-instructie vervangen door de For- methode uit de Parallel FX library. De code die in de for-instructie zelf staat, moet als delegate-functie in één van de argumenten worden meegegeven. Het resultaat is te vinden in codevoorbeeld 2.

Om het effect van het parallelle framework aan te tonen, moeten beide applicaties op dezelfde machine met twee processors draaien. Het resultaat hiervan is te vinden in afbeelding 1. De resultaten geven aan dat de parallelle applicatie bijna twee keer zo snel is. Mocht je over een machine beschikken die vier

```
public static void CodeVoorbeeld2()
{
   DateTime start = DateTime.Now;

   // Doorloop een processor intensieve instructie 200 keer,
   // maar dan parallel zodat alle processors gebruikt worden.
   Parallel.For(0, 200, delegate(int i)
   {
      ProcessorIntensieveInstructie(i);
   });

   Console.WriteLine("Parallel: {0:N0} ms",
      DateTime.Now.Subtract(start).TotalMilliseconds);
}
```

Codevoorbeeld 2



Afbeelding 1. Het aantonen van het effect van het parallelle framework

```
var fibonacci = new int[20];     // array om Fibonacci reeks op te slaan

// Initialiseren Fibonacci reeks
fibonacci[0] = 0;
fibonacci[1] = 1;

// Bereken de overige fibonacci elementen parallel
Parallel.For(2, 20, delegate(int i)
{
   // Zorg voor parallelle uitvoering ook op single-cpu
   // configuraties en voor kleine iteraties
   Thread.Sleep(50);

   // Bereken het volgende element uit de fibonacci reeks
   fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
});

// Print de elementen in de fibonacci reeks naar de Console window
foreach (var i in fibonacci) { Console.WriteLine("{0} ", i); };
```

Codevoorbeeld 3

processors heeft, dan zou de applicatie zelfs bijna vier keer zo snel zijn geweest.

Achterliggende theorie

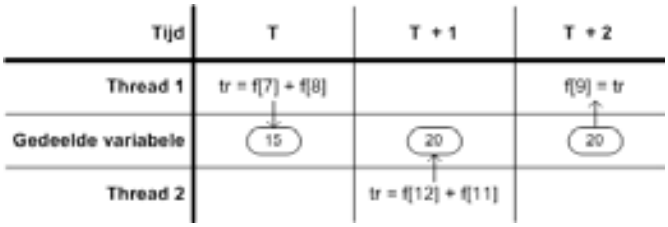
Als je een methode aanroept uit het Parallel FX framework, dan maakt het framework een aantal threads in de threadpool aan. Het aantal threads dat het framework aanmaakt, is afhankelijk van het aantal processors, maar ook criteria als het aantal iteraties in de loop, het aantal threads dat wacht op user-input en het aantal threads dat in slaap is, spelen een rol. De iteraties uit de for-loop worden vervolgens verdeeld over de threads in de threadpool. Mocht een thread eerder klaar zijn dan de andere threads, dan wordt het werk herverdeeld, zodat de loop weer parallel verloopt. Dit wordt ook wel 'work stealing' genoemd. Doordat het framework wat extra werk nodig heeft om het aantal threads te bepalen en aan te maken, zal op de traditionele machines met één processor de niet-parallelle variant iets beter presteren, doordat de parallelle variant een stukje extra overhead heeft. Dit overheadverlies is echter verwaarloosbaar ten opzichte van de totale uitvoertijd. Moet ik nu al mijn for-loops parallel maken? Het antwoord op deze vraag is nee. De beste kandidaten om dit framework toe te passen zijn de 'embarrassingly parallel problems'. Dit zijn problemen die onafhankelijk van elkaar kunnen worden opgelost. Voorbeeld hiervan is het berekenen van pixels in een 3D-projectie. Hier is de Ray Tracer (http://blogs.msdn.com/lukeh/archive/2007/04/03/a-ray-tracer-in-c-3-0.aspx) een implementatie van. Voor niet-onafhankelijke problemen kunnen allerlei concurrency-problemen optreden.

Races

In codevoorbeeld 2 had elke iteratie alleen betrekking op één element uit de array. Je kunt dat uitbreiden naar een rij van Fibonacci van twintig elementen (elk element van de array is de som van de twee voorgaande elementen, de rij begint met 0 en 1). Als je dit algoritme niet parallel uitvoert, dan krijg je een volgende rij te zien:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

Codevoorbeeld 3 toont de parallelle variant van het Fibonacci-algoritme. Bij uitvoering van deze code is de volgende reeks te zien:
0 1 1 2 3 5 8 13 21 34 0 0 0 0 0 0 0 0 0 0

Deze reeks is ontstaan doordat het framework het werk heeft verdeeld over twee threads. Het framework heeft de eerste tien



Afbeelding 2. Een schematische weergave hoe meerdere threads omgaan met de gedeelde variabele

iteraties aan de eerste thread toegewezen en de laatste tien iteraties aan de tweede thread. Als de tweede thread in zijn eerste iteratie het elfde element uit de reeks gaat berekenen, telt de thread het negende en het tiende element bij elkaar op. De eerste thread heeft deze elementen nog niet berekend, zodat beide elementen in de array 0 bevatten.

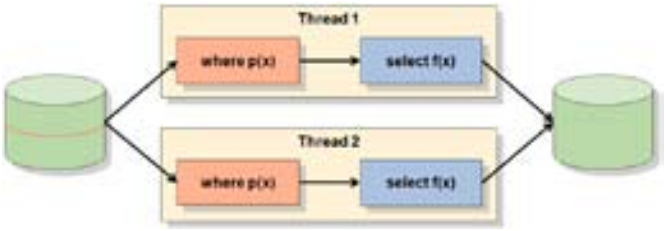
Locks

Als jouw code in meerdere threads verloopt, dan is er nog een valkuil waar je in kunt stappen. Bij het werken met variabelen die door meerdere threads worden gedeeld, kun je onverwachte resultaten te zien krijgen. Laten we het Fibonacci-algoritme wat aanpassen, zodat de threads een variabele delen die het tussenresultaat opslaat.

Een mogelijk resultaat van het uitvoeren van de code in voorbeeld 4 levert een volgende reeks op:
0 1 1 2 3 5 0 5 10 20 5 5 15 0 15 15 30 45 15 15
Om dit gedrag te verklaren moeten we kijken hoe de code uit het voorbeeld op de processor wordt uitgevoerd. De processor verlangt namelijk geen C#-code, maar assembler-instructies. Voor zowel het berekenen van het tussenresultaat als voor het opslaan van het tussenresultaat in de array zijn meerdere assembler-instructies nodig. Het OS kan na elke assembler-instructie een andere thread actief maken. Om er voor te zorgen dat in dit kleine voorbeeld dit gedrag wordt gesimuleerd, is een Sleep-commando toegevoegd die een andere thread actief maakt. Afbeelding 2 toont schematisch hoe meerdere threads omgaan met de gedeelde variabele.

```
int tussenresultaat;// Gedeelde variabele

var fibonacci = new int[20];    // array om Fibonacci reeks op te slaan

// Initialiseren Fibonacci reeks
fibonacci[0] = 0;
fibonacci[1] = 1;

// Bereken de overige fibonacci elementen parallel
Parallel.For(2, 20, delegate(int i)
{
   // Bereken het volgende element uit de fibonacci reeks
   tussenresultaat = fibonacci[i - 1] + fibonacci[i - 2];

   // Zorg voor parallelle uitvoering ook op single-cpu
   // configuraties en voor kleine iteraties
   Thread.Sleep(50);

   // Sla het tussenresultaat op in de reeks
   fibonacci[i] = tussenresultaat;
});

// Print de elementen in de fibonacci reeks naar de Console window
foreach (var i in fibonacci) { Console.WriteLine("{0} ", i); };
```

Codevoorbeeld 4



Afbeelding 3. Een voorbeeld van partioning

De eerste thread berekent de nieuwe waarde voor het negende element in de Fibonacci-rij en slaat dit op in de gedeelde variabele 'tussenresultaat'. Het OS maakt vervolgens de tweede thread actief. De tweede thread telt het elfde en twaalfde element uit de Fibonacci-rij op en overschrijft de gedeelde variabele met het resultaat hiervan. Vervolgens wordt de eerste thread door het OS weer actief gemaakt. Deze thread vult het negende element in de rij met de waarde die in de gedeelde variabele staat. De thread gebruikt hiervoor de waarde 20 in plaats van de verwachte waarde 15. Je kunt in C# dit probleem voorkomen door kritische secties te benoemen. Dit zijn secties waar slechts één thread tegelijkertijd code aan het uitvoeren is. Codevoorbeeld 5 toont hoe we het locking-probleem in het Fibonacci-algoritme thread-safe kunnen maken.

Met het commando lock kun je aangeven dat een aantal instructies kritisch is. Een lock registreer je op een object. Een thread kan alleen de sectie binnengaan als geen andere thread een lock heeft aangevraagd voor hetzelfde object. Omdat het race-probleem hiermee niet is opgelost, kan het zijn dat het resultaat niet de correcte Fibonacci-rij toont.

PLINQ

Het is met de Parallel FX Library niet alleen mogelijk om loops parallel uit te voeren. Ook LINQ- statements kun je met een kleine aanpassing parallel laten uitvoeren. Codevoorbeeld 6 laat zien hoe je van een LINQ-query een PLINQ-query kunt maken. Door enkel

```
int tussenresultaat;          // Gedeelde variabele
var SyncObject = new object(); // Object die de kritische sectie beheert

var fibonacci = new int[20];     // array om Fibonacci reeks op te slaan

// Initialiseer Fibonacci reeks
fibonacci[0] = 0;
fibonacci[1] = 1;

// Bereken de overige fibonacci elementen parallel
Parallel.For(2, 20, delegate(int i)
{
  // Start van de kritische sectie
  lock (SyncObject)
  {
    // Bereken het volgende element uit de fibonacci reeks
    tussenresultaat = fibonacci[i - 1] + fibonacci[i - 2];

    // Zorg voor parallelle uitvoering ook op single-cpu
    // configuraties en voor kleine iteraties
    Thread.Sleep(50);

    // Sla het tussenresultaat op in de reeks
    fibonacci[i] = tussenresultaat;
  }
});

// Print de elementen in de fibonacci reeks naar de Console window
foreach (var i in fibonacci) { Console.WriteLine("{0} ", i); };
```

Codevoorbeeld 5.

```
IEnumarable<T> data = ......;   // Databron, wat van IEnumerable<T>
erft

// p(x) is het predicaat (zoals bv "x < 100")
// f(x) is de functie op x (zoals bv "x ^ 2")

// Gebruik LINQ om de databron te query-en
var q = from x in data where p(x) select f(x);

// Laat de query parallel uitvoeren
var q = from x in data.AsParallel where p(x) select f(x);
```

Codevoorbeeld 6

het toevoegen van de extensie AsParallel op de databron heb je gerealiseerd dat de query parallel kan worden uitgevoerd.

Deze aanpassing heeft enkel zin op databronnen die zelf niet parallel zijn opgezet. Heb je een LINQ- query waarvan de databron een database is, dan wordt het uitvoeren van de LINQ-query parallel door de database afgehandeld. Echter bij het gebruik van XML of objecten als databron voor de LINQ- query is dat niet het geval en is PLINQ een waardevolle toepassing.

Achterliggende theorie

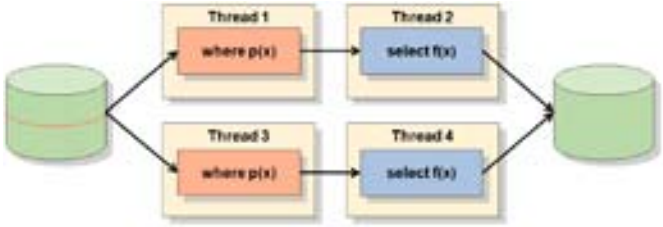
Als je de AsParallel-extensie gebruikt, dan kan het framework op de achtergrond de query gaan opdelen door middel van partitioning of door pipelining. Hoe het framework een keuze maakt, is afhankelijk van allerlei factoren en valt buiten de scope van dit artikel. Afbeelding 3 toont partitioning, het verdelen van de databron uit codevoorbeeld 6 over meerdere threads.

Met pipelining worden tussenresultaten doorgeschoven van de ene naar de andere thread. Thread 1 haalt uit de databron 'data' de records die voldoen aan p(x). Thread 2 past de functie f(x) toe en voegt het resultaat uit deze functie toe aan het query-resultaat. Deze techniek is schematisch te zien in afbeelding 4.

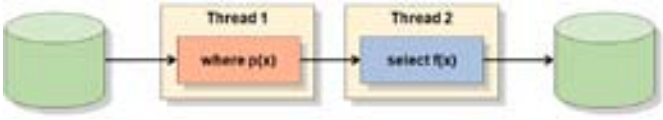
Als je beide technieken combineert, dan kan het framework het werk verdelen over nog meer threads, zodat machines met vier, acht of nog meer processors optimaal benut kan worden. Deze combinatie is schematisch te zien in afbeelding 5.

Met zorg inzetten

Tot nu toe is het maken van multi-threaded applicaties lastig en foutgevoelig. Door het gebruik van Parallel FX Library krijgt de gebruiker een krachtig middel in handen om code parallel uit te laten voeren. Je kunt met eenvoudige aanpassingen een multi-threaded applicatie maken, zodat een applicatie van alle processors op een multi-core computer gebruik kan maken. Er zijn wel kanttekeningen bij het gebruik van dit framework. Een loop parallel



Afbeelding 4. Functie f(x) wordt toegepast door thread 2



Afbeelding 5. Het framework verdeelt het werk over meer threads

laten verlopen, zal de applicatie niet altijd sneller maken. Ook introduceer je met het gebruik van parallelle loops de standaard problemen van een multi-threaded applicatie. De gebruikers van Linq-To-Xml-instructies en Linq-To-Objects-instructies, die veel processortijd nodig hebben, zullen baat hebben bij PLINQ. Maar ook voor PLINQ geldt dat de ontwikkelaar er niet zonder meer van uit kan gaan dat het inzetten van deze techniek een versnelling van de applicatie betekent. Als het framework met zorg wordt inzet, dan is de Parallel FX Library een goede aanvulling op de gereedschapskist van de ontwikkelaar.

Ewald Hofman is als Senior Consultant werkzaam bij Avanade (www.avanade.com), een samenwerkingsverband tussen Microsoft en Accenture. Voor vragen of opmerkingen is hij te bereiken op ewald.hofman@avanade.com.

Referenties

Download zelf de Parallel Extensions, December 2007 CTP: http://www.microsoft.com/downloads/details.aspx?familyid=E848DC1D-5BE3-4941-8705-024BC7F180BA

Het Developer Center voor Parallel Computing: http://msdn2.microsoft.com/en-us/concurrency/default.aspx

Twee artikelen in het MSDN magazine over parallelle iteraties en PLINQ: http://msdn.microsoft.com/msdnmag/issues/07/10/Futures/default.aspx http://msdn.microsoft.com/msdnmag/issues/07/10/PLINQ/default.aspx

Blog: Parallel Programming in .NET : http://blogs.msdn.com/pfxteam/