

SPRAWOZDANIE

Równoległość oparta na procesach

Programowanie równoległe

17.01.2015

Karol Suwalski 125NCI B

<https://github.com/SuwalskiKarol/oor.git>

Wstęp

Po przeanalizowaniu dziesiątek for czy książek jak np. „C# Multithreaded and Parallel Programming” Autorstwa Rodney’a Ringler’a ja dalej nie rozumiem czym jest multiprocessing z punktu widzenia programowania :D. Takie słowa jak multithreading, multiprocessing czy miltiple core są zazwyczaj na forach wymieniane naprzemiennie. Ludzie nie czają czym się różni wątek od procesu. Podobno ludzie oddawali Panu doktorowi tę laborkę w C#. Ja nie wiem co oni oddawali, pewnie Taski albo background workery. Nie wiem, mam blokadę umysłową na tym punkcie xD.

Niestety muszę zaliczyć tę laborkę bo mi wybór się trochę skończył. Także zaprezentuję kolejną technologię wprowadzoną przez Microsoft pozwalającą na równoległe wykonywanie kodu.

Czym jest LINQ(Language Integrated Query)?

Pozwala nam na szybkie i skuteczne tworzenie zapytań do obiektów. Ponieważ wszystko w języku C# jest obiektem, dlatego możemy o te obiekty pytać pisząc proste zapytania i korzystać z metod rozszerzających działających na tych obiektach.

Tworzenie zapytań LINQ pozwala na wydzielenie pewnej części obiektów według naszych potrzeb tak samo jak dzieje się to z danymi w SQLu. Wprowadzamy warunki a następnie operujemy na kolekcjach obiektów.

Prościutki przykład sortujący liczby:

```
staticvoid Main(string[] args)
{
    int[] kolekcjaLiczb = new[] { 1, 4, 6, 5, 3, 2, 8, 9, 0 };
    var query2 = kolekcjaLiczb.Select(n => n).ToArray().OrderBy(b => b);
    foreach (var item in query2)
    {
        Console.Write(item + " ");
    }
    Console.ReadLine();
}
```

Co to jest PLINQ?

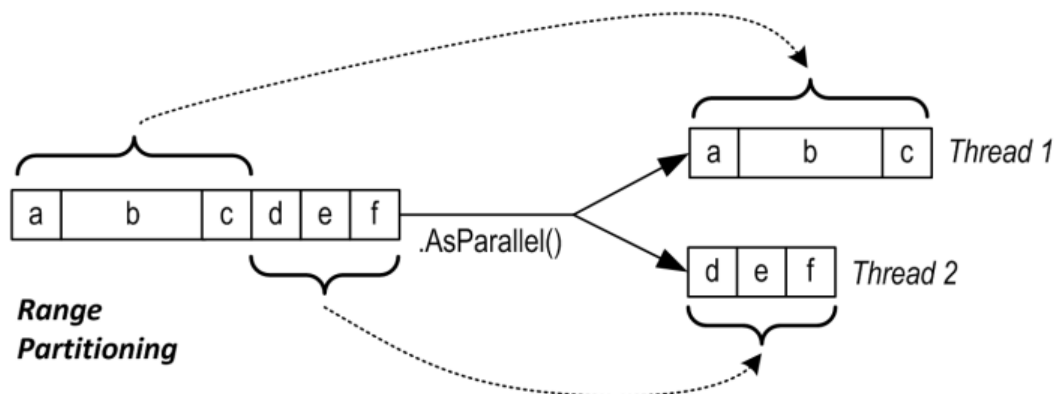
PLINQ to skrót od ParallelLinq, czyli są to zapytania wykonywane równoległe. Samodzielne pisanie LINQ w sposób równoległy jest dość niewygodne i dlatego Microsoft wprowadził PLINQ. Należy oczywiście zawsze pamiętać, że próba zrównoleglenia operacji, które muszą po prostu zostać wykonane sekwencyjnie, zwykle pogarsza wydajność.

Z tego wynika fakt, że możemy zrównoleglić wyłącznie zadania, które da się od siebie oddzielić i można je wykonywać niezależnie od siebie – tzn. zadanie A nie potrzebuje danych z zadania B. Taka sytuacja jest komfortowa i wtedy najwięcej zyskujemy z PLINQ. Kluczem do prawidłowej implementacji PLINQ jest, zatem podział (partition)

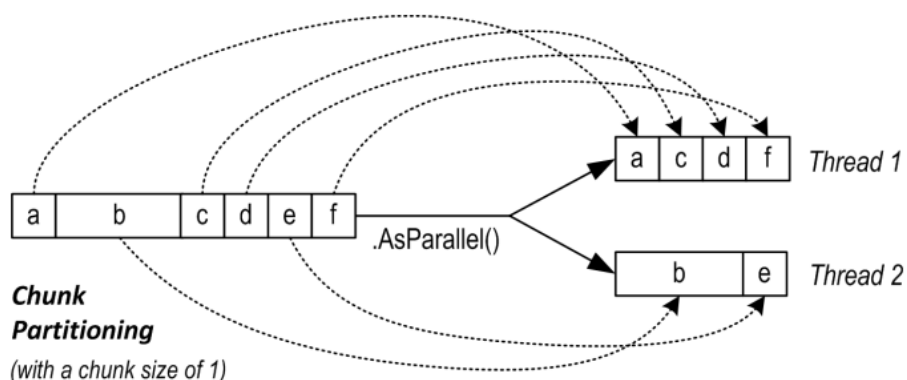
zapytania na części. PLINQ wewnętrznie dokonuje analizy zapytania i na podstawie tego, dokonuje wyboru najbardziej optymalnej strategii.

PLINQ rozdziela zadania na aktualnie dostępne rdzenie. Robi korzystając z 4 metod rozdzielania tablic:

- RangePartitioning - rozdziela równomiernie pomiędzy różne wątki np. elementy od 0 do 5 są przetwarzane przez ThreadA, od 6 do 10 przez Thread B itp. Taki komfort niestety można mieć dla kolekcji z góry określonych, czyli takich, które mają właściwość Length oraz można do nich dostać się przez indeks.



- ChunkPartitioning – Wykorzystywany, jeśli długość nie jest znana. Najpierw przydzielana jest stała liczba elementów do każdego wątku. Po pierwszej iteracji, jeśli nie wszystkie elementy zostały przetworzone, liczba jest podwajana. Szczegóły wewnętrzne zmieniają się, więc nie wiadomo dokładnie czy ThreadA będzie miał najpierw jeden element a potem 2,4,8,16 czy skok jest np. poczwórny. Jedyne co trzeba wiedzieć to fakt, że dla IEnumerable nie można w łatwy sposób dokonać podziału zadań i trzeba wykonywać to heurystycznie.



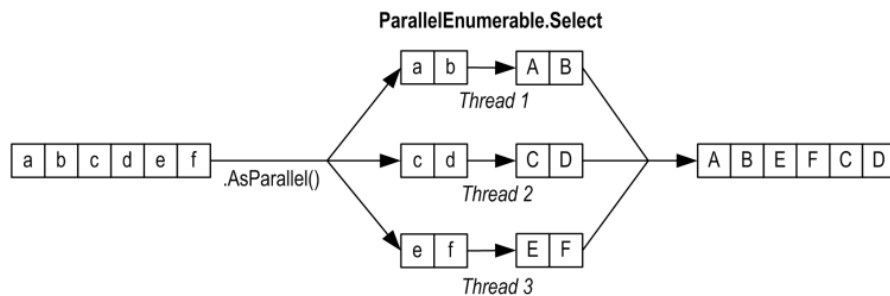
- StripedPartitioning – Wiem tylko tyle, że wykorzystywany dla SkipWhile oraz TakeWhile.

- HashPartitioning – Wykorzystywany dla GroupBy.

PLINQ jest dość sprytny i jeśli uzna, że dane zapytanie nie powinno zostać wykonane w sposób równoległy to zostanie ono przetworzone w klasyczny sposób.

Zapytania PLINQ, podobnie zapytania sekwencyjne LINQ, działają na wszystkich źródłach danych w pamięci `IEnumerable` lub `IEnumerable<T>`.

Działanie podziału danych:



```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ToArray()
```

W jaki sposób PLINQ scala pod koniec podzielone dane wykonywane na różnych wątkach?

Metoda `foreach`. Najpierw dzielimy dane na podgrupy, które później będą wykonywane niezależnie w różnych wątkach. Następnie wywołujemy `ForEach`, który ma charakter sekwencyjny i zawsze wymaga wykonania całości. Jeśli korzystamy z `ForEach`, zawsze musimy poczekać, aż wszystkie wątki wykonają się i na końcu trzeba scalać wynik – dość czasochłonne.

Metoda `forall`. Przetwarza dane, zanim jeszcze wszystkie wątki skończą zadanie. W ten sposób unikamy ostatniego kroku – scalenia poszczególnych zapytań.

```
int[] numbers = Enumerable.Range(1, 50).ToArray();
numbers.AsParallel().Where(n=>n>2).ForAll(Console.WriteLine);
```

Manualne określenie metody scalania za pomocą metody `ParallelMergeOptions`.

Przyjmuję ona następujące wartości :

- `AutoBuffered` – dokonuje buforowania elementów. W praktyce to oznacza, że gdy mamy `ForEach`, elementy dostępne będą dopiero po jakimś czasie, gdy bufor zapełni się.
- `FullyBuffered` – elementy będą dopiero dostępne gdy wszystkie wątki zostaną wykonane. W międzyczasie będą one buforować dane a na końcu bufor zostanie w całości zwrócony.
- `NotBuffered` – brak bufora. Elementy będą wyświetlane natychmiast gdy są tylko dostępne.
- `Default` – aktualnie jest to `AutoBuffered`.

```
int[] numbers = Enumerable.Range(1, 50).ToArray();
var query =
numbers.AsParallel().WithMergeOptions(ParallelMergeOptions.FullyBuffered).Where(n =>
n > 2);
```

Przykład programu wykorzystującego PLINQ

```
Processing..
Time: 0000639

Parallel Processing..
Number of cores: 1 Time: 0033243
Number of cores: 2 Time: 0000032

Processing and calling .ToList()
Number of cores: 1 Time: 1417277
Number of cores: 2 Time: 2087298

Processing and calling .ToList() after PLINQ execution
Number of cores: 1 Time: 1221378
Number of cores: 2 Time: 1125894
```

Step 1

Kod odpowiadający za wykonanie operacji przy użyciu LINQ, czyli sekwencyjnie z wykorzystaniem jednego rdzenia.

```
        for (var i = 1; i <= 1; i++)
        {
var myRange = Enumerable.Range(1, 1000000);

Console.WriteLine("Processing..");

var stopwatch = Stopwatch.StartNew();

var result = myRange.Select(x => x);

stopwatch.Stop();

Console.WriteLine("Time: {0:FFFFFFF}", stopwatch.Elapsed);

myRange = null;
result = null;
        }
```

Step 2

Kod pokazujący wykonanie operacji z wykorzystaniem PLINQ. Słowo `WithDegreeOfParallelism()` określa ile rdzeni ma być użyte do wykonania operacji. Użyłem pętli, aby użył tyle rdzeni ile jest to możliwe. Równie dobrze moglibyśmy użyć cyfry odpowiadającej ilości rdzeni.

```
Console.WriteLine("Parallel Processing..");

for (var i = 1; i <= Environment.ProcessorCount; i++)
{
var myRange = Enumerable.Range(1, 1000000);

var stopwatch = Stopwatch.StartNew();

var result = myRange.AsParallel()
                    .WithDegreeOfParallelism(i)
                    .Select(x => x);

stopwatch.Stop();

Console.WriteLine("Number of cores: {0} Time: {1:FFFFFFF}", i, stopwatch.Elapsed);

myRange = null;
```

Step 3

Kod pokazujący, użycie słowa `toList()`. Metoda ta zmienia zapytanie z `IEnumerable<T>` na listę, powoduje to sprawniejszy podział zapytania i lepsze wykorzystanie rdzeni, co w zależności od przypadku może przyspieszyć działanie kodu.

```
Console.WriteLine("Processing and calling .ToList()");

for (var i = 1; i <= Environment.ProcessorCount; i++)
{
    var myRange = Enumerable.Range(1, 1000000);

    var stopwatch = Stopwatch.StartNew();

    var result = myRange.AsParallel()
        .WithDegreeOfParallelism(i)
        .Select(x => x).ToList();

    stopwatch.Stop();

    Console.WriteLine("Number of cores: {0} Time: {1:FFFFFFF}", i,
        stopwatch.Elapsed);

    myRange = null;
    result = null;
}
```

Step 4

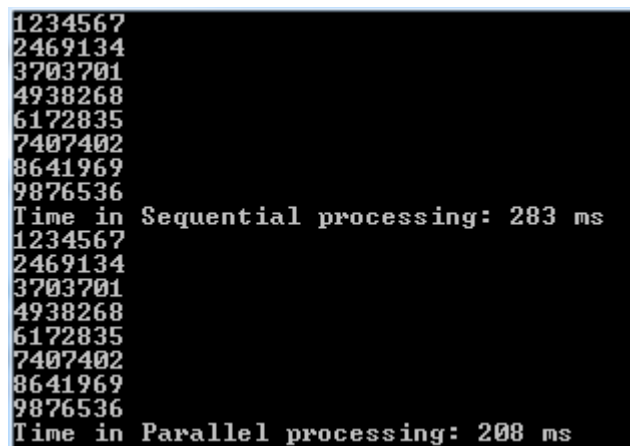
Kod jest podobny do poprzedniego z tą różnicą, że słowo `toList()` używamy po zakończeniu operacji PLINQ.

```
var result = myRange.AsParallel()
    .WithDegreeOfParallelism(i)
    .Select(x => x);

result.ToList();
```

Jak już pisałem użycie równoległości nie zawsze się opłaca. Jak widać na powyższym przykładzie sekwencyjnie program wykonuje się szybciej.

Podaję przykład, gdzie użycie równoległości jest opłacalne.



```
1234567
2469134
3703701
4938268
6172835
7407402
8641969
9876536
Time in Sequential processing: 283 ms
1234567
2469134
3703701
4938268
6172835
7407402
8641969
9876536
Time in Parallel processing: 208 ms
```