Технология PLINQ

Технология PLINQ (Parallel LINQ) позволяет автоматически распараллеливать LINQ-запросы для обработки локальных структур данных.

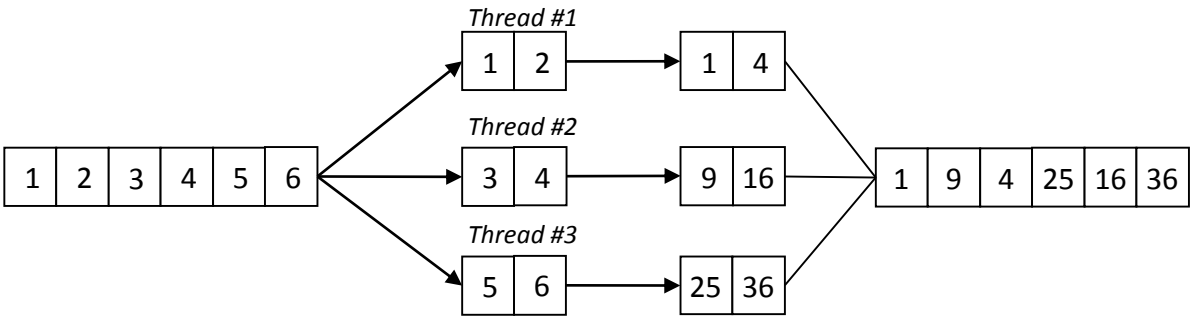
```
IEnumerable<int> numbers = Enumerable.Range(1, 10000);
// Последовательный запрос
var seqQ = from n in numbers
           where n % 2 == 0
           select Math.Pow(n, 2);
// Объявляем запрос, который выполняется параллельно
var parQ = from n in numbers.AsParallel()
          where n % 2 == 0
          select Math.Pow(n, 2);
```

Метод AsParallel преобразуют исходную последовательность типа IEnumerable<TSource> в последовательность типа ParallelQuery<TSource>. Этот тип содержит методы-расширения с теми же именами, как и тип IEnumeralbe<T>, но предполагающие возможное параллельное исполнение на многопроцессорной системе. Другой способ выполнения параллельного запроса связан с использованием объекта ParallelEnumerable, который позволяет сформировать диапазон аналогично объекту Enumerable:

```
// Альтернативный вызов параллельных запросов
var parQ2 = from n in ParallelEnumerable.Range(1, 1000);
                  where n % 2 == 0
                  select Math.Pow(n, 2);
```

Основные этапы выполнения PLINQ-запроса: разделение данных по рабочим потокам, параллельное исполнение запросов на каждом потоке, агрегирование результатов в конечную последовательность.

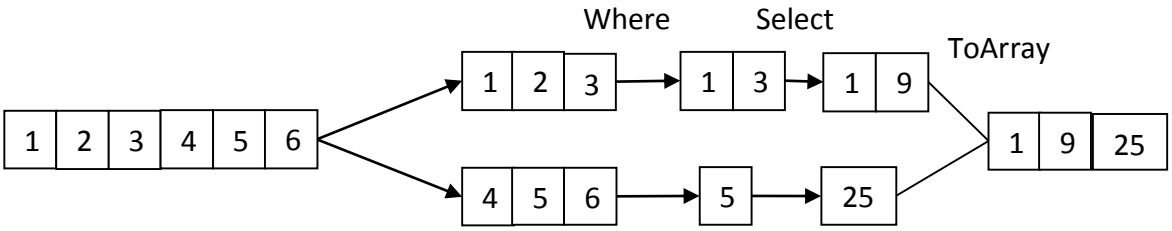
```
numbers.AsParallel().Select(n => Math.Pow(n, 2)).ToArray()
```



Удобство PLINQ заключается в том, что выполняя несколько запросов к последовательности, разделение элементов по потокам осуществляется по возможности вначале обработки. Элементы, попавшие в один поток при выполнении первого оператора, продолжают обрабатываться в этом потоке.

На рисунке изображено выполнение запроса

ParallelEnumerable.Range(1, 6).Where(n => n % 2 != 0).Select(n => n * n).ToArray();



Как и в случае шаблонов класса Parallel, для обработки элементов используются не только рабочие потоки пула, но и в первую очередь пользовательский поток, в котором осуществляется вызов запроса.

```
var threads =
from n in ParallelEnumerable.Range(1, 1000)
     select Thread.CurrentThread.ManagedThreadId;
```

В этом запросе сохраняем номер потока, в котором происходит обработка элементов. Получаем, что половина элементов обрабатывается в основном потоке, другая половина в рабочем потоке пула.

Эффективность распараллеливания

Распараллеливание выполнения запроса связано с накладными расходами на разделение данных, агрегирование результатов. Многие закономерности при обработке PLINQ-запросов сохраняются: чем больше элементов, тем выше эффективность распараллеливания; чем больше вычислительная сложность обработки элемента, те выше эффективность распараллеливания.

Некоторые запросы обладают низкой эффективностью – последовательные аналоги работают быстрее, чем параллельные. Например, запросы, связанные со сравнением элементов (`GroupBy`, `Distinct`, `Join` и др.) разделяют элементы с помощью хэш-секционирования: перед распараллеливанием для каждого элементы вычисляет хэш-код, элементы с одинаковыми хэш-кодами объединяются для обработки в одном потоке. Сложная процедура разделения значительно снижает эффект распараллеливания дальнейшей обработки. Рассмотрим следующий фрагмент:

```
var threads =
from n in ParallelEnumerable.Range(1, 1000)
     select Thread.CurrentThread.ManagedThreadId;
     var diff = threads.Distinct().ToArray();
     int nThr = diff.Length;
```

Запрос `threads` сохраняет номер потока, в котором осуществляется обработка элемента. Для получения массива уникальных номеров `diff` используем оператор выявления различий `Distinct` и оператор преобразования в массив `ToArray`. Если изменить последовательность операторов `Distinct` и `ToArray`, то получим более производительный код:

```
  var diff = threads.ToArray().Distinct();
```

Быстродействие выполнения второго варианта в 2 – 2.5 раза превышает параллельную версию. Разный порядок операторов приводит к тому, что метод `Distinct` применяется для массива, то есть типа `IEnumerable<T>`, а не для типа `ParallelQuery<T>`. Поэтому вызывается последовательная версия поиска различных элементов.

Запросы `Take, TakeWhile, Skip, SkipWhile` работают только с исходным порядком элементов. Например, запрос `Take(5)` отбирает первые пять элементов. Анализатор PLINQ не распараллеливает такие запросы.

Для обязательного распараллеливания запроса вне зависимости от эффективности применяется модификатор `WithExecutionMode` с параметром `ForceParallelism`:

```
var threads = "abcdef".AsParallel()
     .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
     .Where(c => true)
     .Take(3)
     .Select(c=> Thread.CurrentThread.ManagedThreadId);
```

```
int nThr = threads.ToArray().Distinct().Count();
```

Несмотря на то, что оператор `Where` фактически пропускает все элементы, анализатор PLINQ-запросов по умолчанию не распараллеливал бы выполнение запроса. Но режим `ForceParallelism` позволяет включить распараллеливание, например, в отладочных целях.

Если какая-то часть запроса обязательно должна выполняться последовательно, необходимо выполнить обратное преобразование `ParallelEnumerable` в `IEnumerable` с помощью метода `AsSequential`. Такое преобразование может быть полезным, если в запросе используются *потоконебезопасные* методы, параллельное выполнение которых может привести к ошибкам.

```
var q = data.AsParallel()
.Select(item => DoSome(item)).Where(item => IsValid(item))
.AsSequential().Select(item => DoSomeSeq(item))
.ToArray();
```

Первые два оператора `Select` и `Where` выполняются параллельно. Перед вызовом последнего оператора элементы собираются в одну последовательность.

Буферизация

При выполнении `LINQ`-запросов элементы обрабатываются последовательно в процессе перебора, отсутствует какая-либо буферизация результатов. Технология `PLINQ` предоставляет три режима буферизации.

По умолчанию используется авто-буферизация (`AutoBuffered`) - объем буфера для вычисления результатов определяется исполняющей средой.

```
var numbers = ParallelEnumerable.Range(1, 10);
var query = numbers.Select(n =>
     {
          Thread.Sleep(500);
          Console.WriteLine("Prepare item {0}, thread {1}",
          n, Thread.CurrentThread.ManagedThreadId);
               return n;
     });
foreach(int i in query)
     Console.WriteLine("Got item {0}", i);
```

При обращении к первому элементу запроса осуществляется параллельная обработка части элементов.

```
     Prepare item 1, thread 10
     Prepare item 6, thread 11
     Prepare item 2, thread 10
     Prepare item 7, thread 11
```

```
     Prepare item 3, thread 10
     Prepare item 8, thread 11
     Prepare item 4, thread 10
     Prepare item 9, thread 11
     Prepare item 5, thread 10
     Got item 1
     Got item 2
     Got item 3
     Got item 4
     Got item 5
     Prepare item 10, thread 11
     Got item 6
     Got item 7
     Got item 8
     Got item 9
     Got item 10
```

Так как число элементов достаточно небольшое, то практически все элементы были обработаны при первом обращении к запросу.

Полная буферизация (`fully-buffered`) позволяет выполнить запрос полностью до предоставления результатов вне зависимости от числа элементов.

```
var numbers = ParallelEnumerable.Range(1, 10);
var q = numbers
     .WithMergeOptions(ParallelMergeOptions.FullyBuffered)
     {
          Thread.Sleep(500);
          Console.WriteLine("Prepare item {0}, thread {1}",
          n, Thread.CurrentThread.ManagedThreadId);
          return n;
     });
foreach(int i in query)
     Console.WriteLine("Got item {0}", i);
```

Обращение к первому элементу инициирует параллельную обработку всех элементов:

```
     Prepare item 6, thread 10
     Prepare item 1, thread 11
     Prepare item 7, thread 10
     Prepare item 2, thread 11
     Prepare item 8, thread 10
     Prepare item 3, thread 11
     Prepare item 9, thread 10
     Prepare item 4, thread 11
     Prepare item 10, thread 10
     Prepare item 5, thread 11
     Got item 1
     Got item 2
     Got item 3
     Got item 4
```

```
Got item 5
Got item 6
Got item 7
Got item 8
Got item 9
Got item 10
```

Режим с полной буферизацией обладает худшим временем получения первого элемента, но получение всей результирующей последовательности может занимать меньше времени, чем другие режимы.

Третий режим не использует буферизацию.

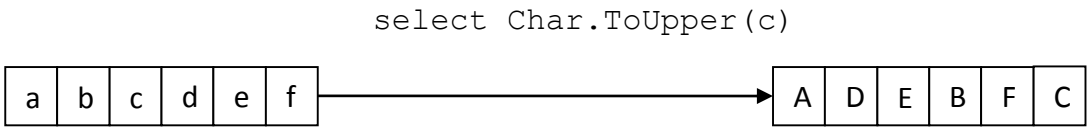
```
var numbers = ParallelEnumerable.Range(1, 10);
var q = numbers
     .WithMergeOptions(ParallelMergeOptions.NotBuffered)
          {
               Thread.Sleep(500);
               Console.WriteLine("Prepare item {0}, thread {1}",
                    n, Thread.CurrentThread.ManagedThreadId);
               return n;
          });
     foreach(int i in query)
          Console.WriteLine("Got item {0}", i);
```

Элементы вычисляются по мере обращения:

```
Prepare item 1, thread 6
Got item 1
Prepare item 6, thread 11
Got item 6
Prepare item 2, thread 6
Got item 2
Prepare item 7, thread 11
Got item 7
Prepare item 3, thread 6
Got item 3
Prepare item 8, thread 11
Got item 8
Prepare item 4, thread 6
Got item 4
Prepare item 9, thread 11
Got item 9
Prepare item 5, thread 6
Got item 5
Prepare item 10, thread 11
Got item 10
```

Порядок элементов

При выполнении параллельных запросов не гарантируется сохранение порядка элементов в результирующей последовательности.



Для гарантированного сохранения порядка необходимо использовать метод `AsOrdered`, который после агрегирования результатов выполняет восстановление порядка.

```
var q = "abcdefgh".AsParallel()
          .Select(c=>Char.ToUpper(c)).AsOrdered().ToArray();
```

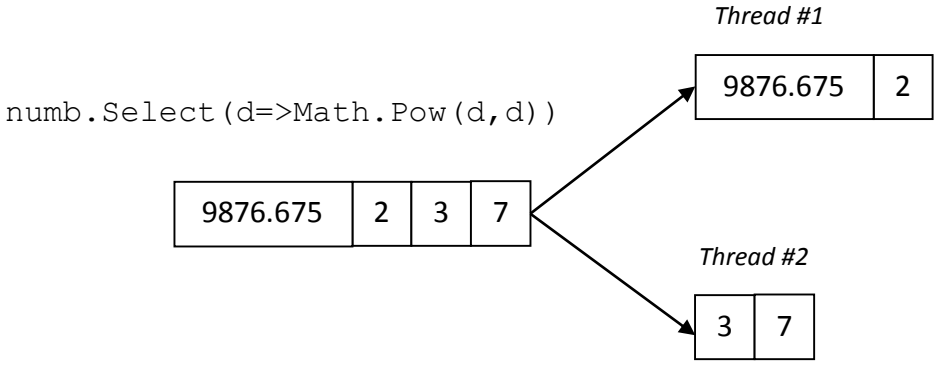
Обязательное упорядочивание результатов приводит к ухудшению эффективности распараллеливания. Модификатор `AsUnordered` позволяет снять требование сохранения порядка и повысить эффективность распараллеливания.

Разделение данных

Разбиение элементов последовательности по потокам осуществляется в соответствии с одной из трех стратегий: блочное разделение (`chunk partitioning`), разделение по диапазону (`range partitioning`), хэш-секционирование (`hash sectioning`).

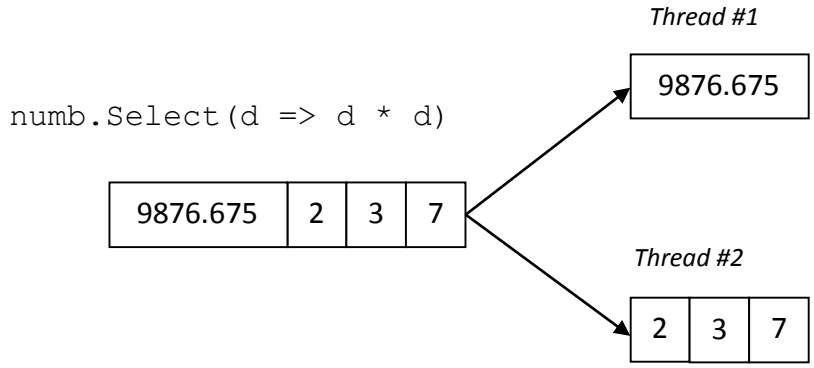
Хэш-секционирование требует расчета хэш-значений для всех элементов последовательности; элементы с одинаковыми хэш-значениями обрабатываются одним и тем же потоком. Хэш-секционирование выполняется для операторов, сравнивающих элементы: `GroupBy`, `Join`, `GroupJoin`, `Intersect`, `Except`, `Union`, `Distinct`.

При разделении по диапазону последовательность разбивается на равное число элементов, каждая порция обрабатывается в одном рабочем потоке. Такое разделение является достаточно эффективным, так как приводит к полной независимости обработки элементов на разных потоках и не требует какой-либо синхронизации. Недостатком такой декомпозиции является несбалансированность загруженности потоков в случае разных вычислительных затрат при обработке элементов последовательности.



Разная вычислительная нагрузка при обработке элементов приводит к несбалансированности загрузки потоков.

Динамическое разделение данных позволяет получить более равномерную загрузку потоков, но требуют синхронизации потоков для доступа к исходной последовательности.



При динамическом (блочном) разделении каждый поток, участвующий в обработке, получает по фиксированной порции элементов (chunk). В качестве порции может быть и один элемент. После обработки своей порции поток обращается за следующей порцией.

По умолчанию при выполнении PLINQ-запросов выполняется разделение по диапазону, кроме запросов, требующих хэш-секционирования. Для выполнения запросов с динамическим разделением необходимо использовать объект `Partitioner`.

```
var parNumbers = ParallelEnumerable.Range(1, 1000);
// Range-partition
var q1  = (from n in parNumbers
          where n % 3 == 0
          select n * n).ToArray();
// Range-partition
double[] ard =
          new double[] {3.4, 56565.634, 7.8, 9.9, 2.4};
var q2 = ard.AsParallel().Select(d =>
Math.Sqrt(d)).ToArray();
// Block-partition
var q3 = Partitioner.Create(ard, true).AsParallel()
               .Select(d=>Math.Sqrt(d)).ToArray();
```

Первый и второй запросы используют разделение по диапазону. Третий запрос использует динамическую декомпозицию. Второй аргумент метода Partitioner.Create задает режим декомпозиции: false – разделение по диапазону, true – динамическая (блочная) декомпозиция.

Обработка исключений

Исключения, которые могут произойти при выполнении PLINQ-запросов, обрабатываются также как и в случае задач с помощью объекта `AggregateException` в блоке `catch`. Блок `try` располагается там, где осуществляется фактический вызов обработки элементов.

```
var q = numbers.Select(n => SomeWork(n)).Where(n =>
          {
               if(n > 0)
                    return true;
               else
                    throw new Exception();
          });
try
{
     foreach(int n in q)
          Console.WriteLine(n);
 }
catch(AggregateException ae)
{
     Console.WriteLine("Some error was happened!");
     return true;
}
```

Обработка элементов начинается при переборе элементов в `foreach`-цикле. Исключение обрабатывается в `catch`-блоке. Объект `AggregateException` содержит список ошибок, возникнувших при обработке элементов, в списке `InnerExceptions`.

Для обработки исключений можно применять методы Flatten и Handle.

Отмена запроса

Для отмены выполнения `PLINQ`-запросов используется объект `CancellationToken`, который передается с помощью метода `WithCancellation`.

```
CancellationTokenSource cts = new CancellationTokenSource();
var q = someData.AsParallel().WithCancellation(cts.Token)
               .Select(d => d * d);

// Задача, которая отменит запрос
Task t = Task.Factory.StartNew(() =>
          {
               Thread.Sleep(100);
               cts.Cancel();
          });

try
{
     var results = data.ToList();
}
catch(OperationCanceledException e)
{
```

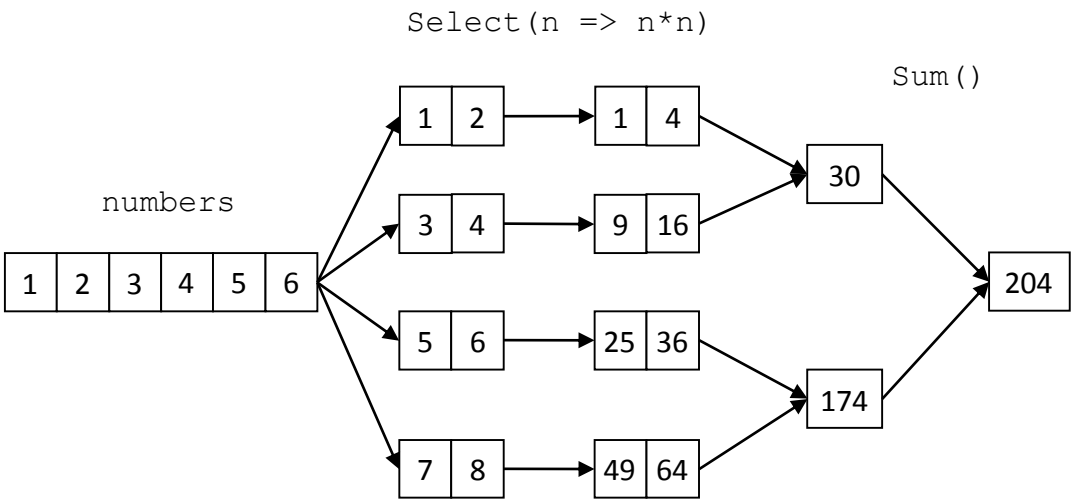
```
     Console.WriteLine("The query was cancelled!");
}
```

В этом фрагменте отмена запроса осуществляется с помощью отдельной задачи. При отмене генерируется исключение `OperationCanceledException`, которое при необходимости можно обработать.

Агрегирование вычислений

Под агрегированием (редукцией) понимается вычисление какого-либо итогового значения для исходного набора элементов. Примером редукции являются: вычисление минимального значения, максимального значения, суммы, произведения и т.д.

Для получения агрегированных результатов можно использовать методы: `Sum`, `Min`, `Max` для вычисления суммы, минимального и максимального значения.



Реализация произвольных агрегированных функций осуществляется с помощью метода `Aggregate`. В следующем фрагменте реализован метод вычисления среднеквадратичного отклонения

```
double CalcStdDevParallel(double[] data)
{
     double mean = data.AsParallel().Average();
     double stdDev = data.AsParallel().Aggregate(
          // Инициализация локальной переменной
          0.0,
          // Вычисления в каждом потоке
          (subtotal, item) =>
               subtotal + Math.Pow(item - mean, 2),
          // Агрегирование локальных значений
          (total, subtotal) =>
               total + subtotal,
          // Итоговое преобразование
          (total) =>
```

```
                Math.Sqrt(total/(data.Length - 1))
    );
return stdDev;
}
```

Первый аргумент – делегат инициализации локальных переменных потоков, который вызывается один раз для каждого потока перед началом вычислений.

```
() => 0.0;
```

Второй аргумент – делегат обработки каждого элемента, принимающий в качестве параметров текущее значение локальной переменной и значение обрабатываемого элемента; возвращается значение локальной переменной. Делегат обработки вызывается для каждого элемента.

```
(local, item) => local + item;
```

Третий аргумент метода агрегирования принимает делегат редукции, определяющий, как частные переменные будут сворачиваться (редуцироваться) в одно итоговое значение. Делегат редукции принимает текущее значение итоговой переменной, локальное значение переменной и возвращает обновленное значение итоговой переменной.

```
(total, local) => total + local;
```

Делегат редукции вызывается столько раз, сколько потоков участвует в обработке метода агрегирования.

Четвертый аргумент – делегат финальной обработки итоговой переменной. Делегат вызывается только один раз после завершения редукции локальных переменных.

```
(total) => Math.Sqrt(total) / (data.Length - 1)
```