Parallel LINQ

Введение

Инфраструктура PLINQ автоматически распараллеливает локальные запросы LINQ. Преимущество PLINQ заключается в простоте использования, т.к. ответственность за выполнение работ по разбиению и объединению результатов перекладывается на .NET Framework.

Для применения PLINQ просто вызовите метод *AsParallel* на входной последовательности и затем продолжайте запрос LINQ обычным образом. Приведенный ниже запрос вычисляет простые числа между 3 и 100 000, обеспечивая полную загрузку всех ядер процессора на целевой машине:

```
// Вычислить простые числа с использованием простого
// (не оптимизированного) алгоритма.

IEnumerable<int> numbers = Enumerable.Range(3, 1000000 - 3);

var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range(2, (int)Math.Sqrt(n))
                  .All(i => n % i > 0)
  select n;

int[] primes = parallelQuery.ToArray();
```

AsParallel представляет собой расширяющий метод в классе *System.Linq.ParallelEnumerable*. Он помещает входные данные в оболочку последовательности, основанной на *ParallelQuery<TSource>*, что вызывает привязку последующих операций запросов LINQ к альтернативному набору расширяющих методов, которые определены в классе *ParallelEnumerable*. Они предоставляют параллельные реализации для всех стандартных операций запросов. По существу они разбивают входную последовательность на порции, которые выполняются в разных потоках, и объединяют результаты снова в единственную

выходную последовательность для дальнейшего потребления (рис. 1).

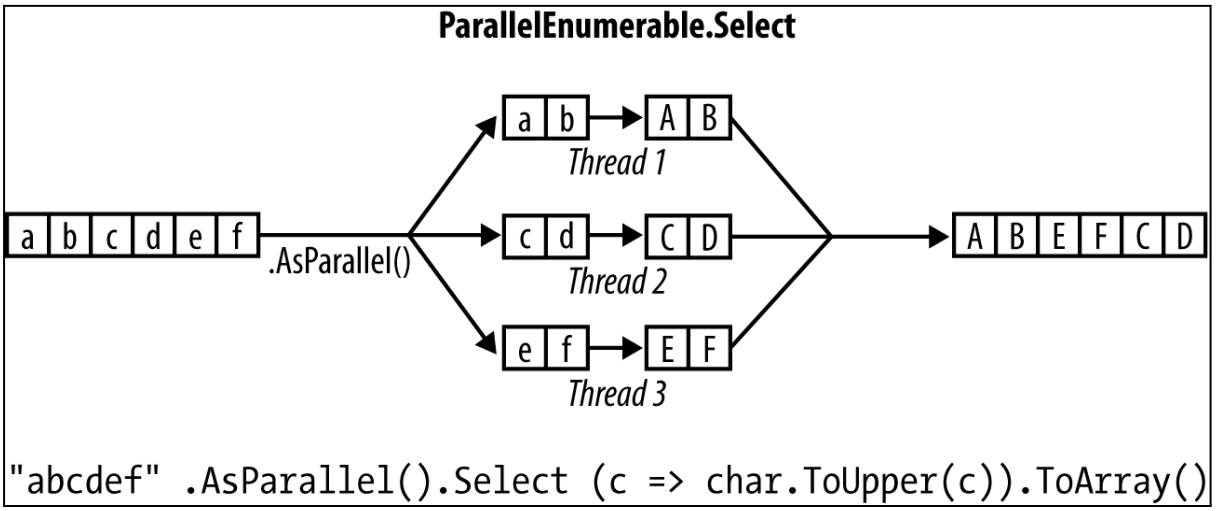

Рис. 1. Модель выполнения PLINQ

Вызов метода *AsSequential* извлекает последовательность из оболочки *ParallelQuery*, так что дальнейшие операции запросов привязываются к стандартному набору операций и выполняются последовательно. Такое действие нужно предпринимать перед вызовом методов, которые имеют побочные эффекты или не являются безопасными в отношении потоков.

Для операций запросов, принимающих две входные последовательности (*Join*, *GroupJoin*, *Concat*, *Union*, *Intersect*, *Except* и *Zip*), метод *AsParallel* должен быть применен к обеим входным последовательностям (иначе сгенерируется исключение).

Однако по мере продвижения запроса применять к нему *AsParallel* нет необходимости, т.к. операции запросов PLINQ выводят еще одну последовательность *ParallelQuery*. На самом деле дополнительный вызов *AsParallel* привносит неэффективность, связанную с тем, что он инициирует слияние и повторное разбиение запроса.

Не все операции запросов можно эффективно распараллеливать. Для операций, не поддающихся распараллеливанию (см. пункт

“Ограничения PLINQ” далее), PLINQ взамен реализует последовательное выполнение. Инфраструктура PLINQ может также оперировать последовательно, если ожидает, что накладные расходы от распараллеливания в действительности замедлят определенный запрос.

Инфраструктура PLINQ предназначена только для локальных коллекций: она не работает с LINQ to SQL или Entity Framework, потому что в таких ситуациях LINQ транслируется в код SQL, который затем выполняется на сервере баз данных. Тем не менее PLINQ можно использовать для выполнения дополнительных локальных запросов в результирующих наборах, полученных из запросов к базам данных.

- Если запрос PLINQ генерирует исключение, то оно повторно генерируется как объект *AggregateException*, свойство *InnerExceptions* которого содержит реальное исключение (или исключения).

Почему метод AsParallel не выбран в качестве варианта по умолчанию?

Учитывая, что метод AsParallel прозрачно распараллеливает запросы LINQ, возникает вопрос: почему в Microsoft решили не распараллеливать стандартные операции запросов, сделав PLINQ вариантом по умолчанию?

Есть несколько причин выбора подхода с включением. Первая причина связана с тем, что для получения пользы от PLINQ в наличии должен быть обоснованный объем работы с интенсивными вычислениями, которую можно было бы поручить рабочим потокам. Большинство потоков LINQ to Objects выполняются очень быстро, и распараллеливание для них не только окажется излишним, но накладные расходы на разбиение, объединение и координацию дополнительных потоков фактически могут даже замедлить их выполнение.
Ниже перечислены другие причины.

- Вывод запроса PLINQ (по умолчанию) может отличаться от вывода запроса LINQ в том, что касается порядка следования элементов (как объясняется в разделе “PLINQ и упорядочивание” далее).
- PLINQ помещает исключения в оболочку *AggregateException* (чтобы учесть возможность генерации множества исключений).
- PLINQ будет давать ненадежные результаты, если запрос вызывает небезопасные к потокам методы.

Наконец, PLINQ предлагает немало способов настройки. Обременение стандартного API-интерфейса LINQ to Objects нюансами такого рода добавило бы путаницы.

Продвижение параллельного выполнения

Подобно обычным запросам LINQ запросы PLINQ оцениваются ленивым образом. Другими словами, выполнение будет инициировано, только когда начнется потребление результатов – как правило, посредством цикла *foreach* (хотя оно также может происходить через операцию преобразования, такую как *ToArray*, или операцию, которая возвращает одиночный элемент либо значение).

Тем не менее при перечислении результатов выполнение продолжается несколько иначе, чем в случае обычного последовательного запроса. Последовательный запрос поддерживается полностью потребителем с применением модели с пассивным источником: каждый элемент из входной последовательности извлекается только тогда, когда он затребован потребителем. Параллельный запрос обычно использует независимые потоки для извлечения элементов из входной последовательности, причем с небольшим упреждением, до того момента, когда они понадобятся потребителю (почти как телесуфлер у дикторов новостей или буфер в проигрывателях компакт-дисков). Затем он обрабатывает элементы параллельно

через цепочку запросов, удерживая результаты в небольшом буфере, чтобы они были готовы при затребовании потребителем. Если потребитель приостанавливает или прекращает перечисление до его завершения, обработчик запроса также приостанавливается или прекращает работу, чтобы не тратить впустую время ЦП или память.

- Поведение буферизации PLINQ можно настраивать, вызывая метод *WithMergeOptions* после *AsParallel*. Стандартное значение *AutoBuffered* перечисления *ParallelMergeOptions* обычно дает наилучшие окончательные результаты. Значение *NotBuffered* отключает буфер и полезно в ситуации, когда результаты необходимо увидеть как можно скорее; значение *FullyBuffered* кеширует целый результирующий набор перед представлением его потребителю (подобным образом изначально работают операции *OrderBy* и *Reverse*, а также операции над элементами, операции агрегирования и операции преобразования).

PLINQ и упорядочивание

Побочный эффект от распараллеливания операций запросов заключается в том, что когда результаты объединены, они необязательно находятся в том же самом порядке, в котором они были получены (см. рис. 1). Другими словами, обычная гарантия предохранения порядка LINQ для последовательностей больше не поддерживается.

Если нужно предохранение порядка, тогда после вызова *AsParallel* понадобится вызвать метод *AsOrdered*:

```
myCollection.AsParallel().AsOrdered()...
```

Вызов метода *AsOrdered* оказывает влияние на производительность, поскольку инфраструктура PLINQ должна отслеживать исходные позиции всех элементов.

Позже последствия от вызова *AsOrdered* в запросе можно отменить, вызвав метод *AsUnordered*: это вводит “точку случайного

тасования”, которая позволяет запросу выполняться более эффективно после ее прохождения. Таким образом, если необходимо предохранить упорядочение входной последовательности только для первых двух операций запросов, то можно поступить так:

```
inputSequence.AsParallel().AsOrdered()
  .QueryOperator1()
  .QueryOperator2()
  .AsUnordered() // Начиная с этой точки,
                 // упорядочивание роли не играет
  .QueryOperator3()
  …
```

Метод *AsOrdered* не является стандартным вариантом, потому что для большинства запросов первоначальное упорядочивание во входной последовательности не имеет значения. Другими словами, если бы метод *AsOrdered* использовался по умолчанию, то к большинству параллельных запросов пришлось бы применять метод *AsUnordered*, чтобы добиться лучших показателей производительности, и поступать так было бы обременительно.

Ограничения PLINQ

Существует несколько практических ограничений относительно того, что инфраструктура PLINQ способна распараллеливать. Ограничения могут быть ослаблены в последующих пакетах обновлений и версиях инфраструктуры .NET Framework.

Следующие операции запросов предотвращают распараллеливание запроса, если только исходные элементы не находятся в своих первоначальных индексных позициях:

- индексированные версии *Select* , *SelectMany* и *ElementAt*.

Большинство операций запросов изменяют индексные позиции элементов (включая операции, удаляющие элементы, такие как *Where*). Это означает, что если нужно использовать

предшествующие операции, то они обычно должны располагаться в начале запроса.

Следующие операции запросов допускают распараллеливание, но применяют затратную стратегию разбиения, которая иногда может оказываться медленнее последовательной обработки:

- *Join* , *GroupBy*, *GroupJoin*, *Distinct* , *Union*, *Intersect* и *Except*.

Перегруженные версии операции *Aggregate*, принимающие начальное значение (в аргументе *seed*), в их стандартном виде не поддерживают возможность распараллеливания – в PLINQ для такой цели предлагаются специальные перегруженные версии (см. пункт “Оптимизация PLINQ” далее).

Все остальные операции поддаются распараллеливанию, хотя их использование не гарантирует, что запрос будет распараллелен. Инфраструктура PLINQ может выполнять запрос последовательно, если ожидает, что накладные расходы от распараллеливания приведут к замедлению имеющегося конкретного запроса. Такое поведение можно переопределить и принудительно применять параллелизм, вызвав показанный ниже метод после *AsParallel*:

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

Пример: параллельная программа проверки орфографии

Предположим, что требуется написать программу проверки орфографии, которая выполняется быстро для очень больших документов за счет использования всех свободных процессорных ядер. Выразив алгоритм в виде запроса LINQ, мы легко можем его распараллелить.

Первый шаг предусматривает загрузку словаря английских слов в объект *HashSet*, чтобы обеспечить эффективный поиск:

```
if (!File.Exists(wordLookupFile))  // Содержит около 150 000 слов
  new WebClient().DownloadFile(
    "http://www.albahari.com/ispell/allwords.txt",
    "WordLookupFile");

var wordLookup = new HashSet<string>(
   File.ReadAllLines("WordLookupFile"),
   StringComparer.InvariantCultureIgnoreCase);
```

Затем мы будем применять полученное средство поиска слов для создания тестового “документа”, содержащего массив из миллиона случайных слов. После построения массива мы внесем пару орфографических ошибок:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range(0, 1000000)
  .Select(i => wordList[random.Next(0, wordList.Length)])
  .ToArray();

wordsToTest[12345] = "woozsh";     // Внесение пары
wordsToTest[23456] = "wubsie";     // орфографических ошибок.
```

Теперь мы можем выполнить параллельную проверку орфографии, сверяя *wordsToTest* с *wordLookup*. PLINQ позволяет делать это очень просто:

```
var query = wordsToTest
  .AsParallel()
  .Select((word, index) =>
            new IndexedWord { Word = word, Index = index })
  .Where(iword => !wordLookup.Contains(iword.Word));
foreach (var mistake in query)
  Console.WriteLine(mistake.Word + " - index = " +
mistake.Index);
// ВЫВОД:
// woozsh - index = 12345
// wubsie - index = 23456
```

IndexedWord – это специальная структура, которая определена следующим образом:

```
struct IndexedWord { public string Word; public int Index; }
```

Метод *wordLookup.Contains* в предикате придает запросу определенный “вес” и делает уместным его распараллеливание.

- Мы могли бы слегка упростить запрос за счет использования анонимного типа вместо структуры *IndexedWord*. Однако это привело бы к снижению производительности, т.к. анонимные типы (будучи классами, а потому ссылочными типами) привносят накладные расходы на выделение памяти в куче и последующую сборку мусора.

- Разница может оказаться недостаточной, чтобы иметь значение в последовательных запросах, но в случае параллельных запросов весьма выгодно отдавать предпочтение выделению памяти в стеке. Причина в том, что выделение памяти в стеке хорошо поддается распараллеливанию (поскольку каждый поток имеет собственный стек), в то время как в противном случае все потоки должны состязаться за одну и ту же кучу, управляемую единственным диспетчером памяти и сборщиком мусора.

Использование *ThreadLocal<T>*

Давайте расширим наш пример, распараллелив само создание случайного тестового списка слов. Мы структурировали его как запрос LINQ, так что все должно быть легко. Вот последовательная версия:

```
string[] wordsToTest = Enumerable.Range(0, 1000000)
  .Select(i => wordList[random.Next(0, wordList.Length)])
  .ToArray();
```

К сожалению, вызов метода *random.Next* небезопасен в отношении потоков, поэтому работа не сводится к простому добавлению в запрос вызова *AsParallel*. Потенциальным решением может быть написание функции, помещающей вызов *random.Next* внутрь блокировки, но это ограничило бы параллелизм. Более удачный вариант предусматривает применение класса *ThreadLocal<Random>* с целью создания отдельного объекта

Random для каждого потока. Тогда распараллелить запрос можно следующим образом:

```
var localRandom = new ThreadLocal<Random>
  (() => new Random(Guid.NewGuid().GetHashCode()));

string[] wordsToTest = Enumerable.Range(0, 1000000)
  .AsParallel()
  .Select(i => wordList[localRandom.Value.Next(0,
wordList.Length)])
  .ToArray();
```

В нашей фабричной функции для создания объекта *Random* мы передаем хеш-код *Guid*, гарантируя тем самым, что даже если два объекта *Random* создаются в рамках короткого промежутка времени, то они все равно будут выдавать отличающиеся последовательности случайных чисел.

Когда необходимо использовать PLINQ?

Довольно заманчиво поискать в существующих приложениях запросы LINQ и поэкспериментировать с их распараллеливанием. Однако обычно это непродуктивно, т.к. большинство задач, для которых LINQ является очевидным лучшим решением, выполняются очень быстро, а потому не выигрывают от распараллеливания. Более удачный подход предполагает поиск узких мест, интенсивно использующих ЦП, и выяснение, могут ли они быть выражены в виде запроса LINQ. (Приятный побочный эффект от такой реструктуризации состоит в том, что LINQ обычно делает код более кратким и читабельным.)

Инфраструктура PLINQ хорошо подходит для естественно параллельных задач. Однако она может быть плохим выбором для обработки изображений, потому что объединение миллионов пикселей в выходную последовательность создаст узкое место. Взамен пиксели лучше записывать прямо в массив или блок неуправляемой памяти и применять класс *Parallel* либо параллелизм задач для управления многопоточностью. (Тем не

менее, объединение результатов можно аннулировать с использованием *ForAll* — мы обсудим данную тему в пункте “Оптимизация PLINQ” далее. Поступать так имеет смысл, если алгоритм обработки изображений естественным образом приспосабливается к LINQ.)

Функциональная чистота

Поскольку PLINQ запускает ваш запрос в параллельных потоках, вы должны избегать выполнения небезопасных к потокам операций. В частности, запись в переменные порождает побочные эффекты, следовательно, она не является безопасной в отношении потоков:

```
// Следующий запрос умножает каждый элемент на его позицию.
// Получив на входе Enumerable.Range(0, 999), он должен
// вывести последовательность квадратов.
int i = 0;
var query = from n in Enumerable.Range(0, 999)
  .AsParallel() select n * i++;
```

Мы могли бы сделать инкрементирование переменной *i* безопасным к потокам за счет применения блокировок, но все еще останется проблема того, что *i* необязательно будет соответствовать позиции входного элемента. И добавление *AsOrdered* в запрос не решит последнюю проблему, т.к. метод *AsOrdered* гарантирует лишь то, что элементы выводятся в порядке, согласованном с порядком, который бы они имели при последовательной обработке – он не осуществляет действительную их обработку последовательным образом.

Взамен данный запрос должен быть переписан с использованием индексированной версии *Select*:

```
var query = Enumerable.Range(0,999)
                      .AsParallel().Select ((n, i) => n * i);
```

Для достижения лучшей производительности любые методы, вызываемые из операций запросов, должны быть безопасными к потокам, не производя запись в поля или свойства (не давать побочные эффекты, т.е. быть функционально чистыми). Если они являются безопасными в отношении потоков благодаря блокированию, тогда потенциал параллелизма запроса будет ограничен продолжительностью действия блокировки, деленной на общее время, которое занимает выполнение данной функции.

Установка степени параллелизма

По умолчанию PLINQ выбирает оптимальную степень параллелизма для задействованного процессора. Ее можно переопределить, вызвав метод *WithDegreeOfParallelism* после *AsParallel*:

```
… AsParallel().WithDegreeOfPallelism(4) ...
```

Примером, когда степень параллелизма может быть увеличена до значения, превышающего количество ядер, является работа с интенсивным вводом-выводом (скажем, загрузка множества веб-страниц за раз). Тем не менее, начиная с версии .NET Framework 4.5, комбинаторы задач и асинхронные функции предлагают аналогично несложное, но более эффективное решение. В отличие от объектов *Task*, инфраструктура PLINQ не способна выполнять работу с интенсивным вводом-выводом без блокирования потоков (и что еще хуже – потоков из пула).

Изменение степени параллелизма

Метод *WithDegreeOfParallelism* можно вызывать только один раз внутри запроса PLINQ. Если его необходимо вызвать снова, то потребуется принудительно инициировать слияние и повторное разбиение запроса, еще раз вызвав метод *AsParallel* внутри запроса:

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism(2)
  .Where(c => !char.IsWhiteSpace(c))
  .AsParallel().WithDegreeOfParallelism(3) // Инициировать
слияние
                                           // и разбиение
  .Select(c => char.ToUpper(c))
```

Отмена

Отменить запрос PLINQ, результаты которого потребляются в цикле *foreach*, легко: нужно просто прекратить цикл *foreach* и запрос будет автоматически отменен по причине неявного освобождения перечислителя.

Отменить запрос, который заканчивается операцией преобразования, операцией над элементами или операцией агрегирования, можно из другого потока через признак отмены. Чтобы вставить такой признак, необходимо после вызова *AsParallel* вызвать метод *WithCancellation*, передав ему свойство *Token* объекта *CancellationTokenSource*. Затем другой поток может вызвать метод *Cancel* на источнике признака, что приведет к генерации исключения *OperationCanceledException* в потребителе запроса:

```
IEnumerable<int> million = Enumerable.Range(3, 1000000);

var cancelSource = new CancellationTokenSource();

var primeNumberQuery =
  from n in
million.AsParallel().WithCancellation(cancelSource.Token)
  where Enumerable.Range(2, (int)Math.Sqrt(n)).All(i => n % i >
0)
  select n;

new Thread(() =>
{
  Thread.Sleep(100);        // Отменить запрос по
  cancelSource.Cancel();    // прошествии 100 мс.
}
           ).Start();
try
```

```
{
  // Начать выполнение запроса:
  int[] primes = primeNumberQuery.ToArray();
  // Мы никогда не попадем сюда, потому
  // что другой поток инициирует отмену.
}
catch (OperationCanceledException)
{
  Console.WriteLine("Query canceled");  // Запрос отменен
}
```

Инфраструктура PLINQ не прекращает потоки вытесняющим образом из-за связанной с этим опасности. Взамен при инициировании отмены она ожидает завершения каждого рабочего потока со своим текущим элементом перед тем, как закончить запрос. Это означает, что любые внешние методы, которые вызывает запрос, будут выполняться до полного завершения.

Оптимизация PLINQ

Оптимизация на выходной стороне

Одно из преимуществ инфраструктуры PLINQ связано с тем, что она удобно объединяет результаты распараллеленной работы в единую выходную последовательность. Однако иногда все, что в итоге делается с такой последовательностью – выполнение некоторой функции над каждым элементом:

```
foreach (int n in parallelQuery)
  DoSomething(n);
```

В таком случае, если порядок обработки элементов не волнует, тогда эффективность можно улучшить с помощью метода *ForAll* из PLINQ.

Метод *ForAll* запускает делегат для каждого выходного элемента *ParallelQuery*. Он проникает прямо внутрь PLINQ, обходя шаги

объединения и перечисления результатов. Ниже приведен простейший пример:

```
"abcdef".AsParallel().Select (c => char.ToUpper(c))
        .ForAll (Console.Write);
```

Процесс продемонстрирован на рис. 2.

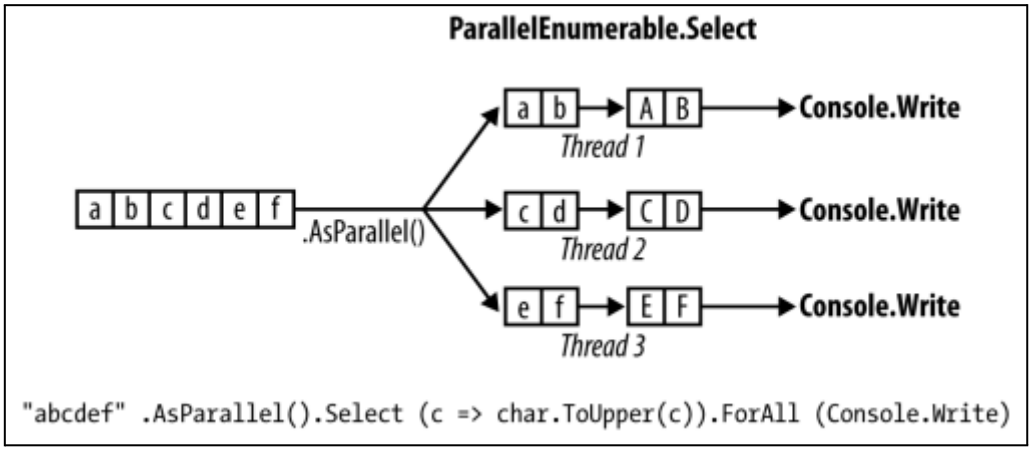

Рис. 2. Метод *ForAll* из PLINQ

- Объединение и перечисление результатов — не массовая затратная операция, поэтому оптимизация с помощью *ForAll* дает наибольшую выгоду при наличии большого количества быстро обрабатываемых входных элементов.

Оптимизация на входной стороне

Для назначения входных элементов потокам в PLINQ поддерживаются три стратегии разбиения:

Стратегия	Распределение элементов	Относительная производительность
Разбиение на основе порций	Динамическое	Средняя
Разбиение на основе диапазонов	Статическое	От низкой до очень высокой
Разбиение на основе хеш-кодов	Статическое	Низкая

Для операций запросов, которые требуют сравнения элементов (*GroupBy*, *Join*, *GroupJoin*, *Intersect* , *Except*, *Union* и *Distinct)* , выбор отсутствует: PLINQ всегда использует разбиение на основе хеш-кодов. Разбиение на основе хеш-кодов относительно неэффективно в том, что оно требует предварительного вычисления хеш-кода каждого элемента (а потому элементы с одинаковыми хеш-кодами могут обрабатываться в одном и том же потоке). Если вы сочтете это слишком медленным, то единственно доступным вариантом будет вызов метода *AsSequential* с целью отключения распараллеливания.

Для всех остальных операций запросов имеется выбор между разбиением на основе диапазонов и разбиением на основе порций. По умолчанию:

- если входная последовательность индексируема (т.е. является массивом или реализует интерфейс *IList<T>*), тогда PLINQ выбирает разбиение на основе диапазонов;
- иначе PLINQ выбирает разбиение на основе порций.

По своей сути разбиение на основе диапазонов выполняется быстрее с длинными последовательностями, для которых каждый элемент требует сходного объема времени ЦП на обработку. В противном случае разбиение на основе порций обычно быстрее.

Чтобы принудительно применить разбиение на основе диапазонов, выполните такие действия:

- если запрос начинается с вызова метода *Enumerable.Range*, то замените его вызовом *ParallelEnumerable.Range*;
- иначе просто вызовите метод *ToList* или *ToArray* на входной последовательности (очевидно, это повлияет на производительность, что также должно приниматься во внимание).

❖ Метод *ParallelEnumerable.Range* – не просто сокращение для вызова *Enumerable.Range(...).AsParallel()*. Он изменяет производительность запроса, активизируя разбиение на основе диапазонов.

Чтобы принудительно применить разбиение на основе порций, необходимо поместить входную последовательность в вызов *Partitioner.Create* (из пространства имен *System.Collection.Concurrent*) следующим образом:

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };

var parallelQuery = Partitioner.Create(numbers, true)
  .AsParallel()
  .Where(...)
```

Второй аргумент *Partitioner.Create* указывает на то, что для запроса требуется балансировка загрузки, которая представляет собой еще один способ сообщения о выборе разбиения на основе порций.

Разбиение на основе порций работает путем предоставления каждому рабочему потоку возможности периодически захватывать из входной последовательности небольшие “порции” элементов с целью их обработки (рис. 3). Инфраструктура PLINQ начинает с выделения очень маленьких порций (один или два элемента за раз) и затем по мере продвижения запроса увеличивает размер порции: это гарантирует, что небольшие последовательности будут эффективно распараллеливаться, а крупные последовательности не приведут к чрезмерным циклам полного обмена. Если рабочий поток получает “простые” элементы (которые обрабатываются

быстро), то в конечном итоге он сможет получить больше порций. Такая система сохраняет каждый поток одинаково занятым (а процессорные ядра “сбалансированными”); единственный недостаток состоит в том, что извлечение элементов из разделяемой входной последовательности требует синхронизации (обычно монопольной блокировки) – и в результате могут появиться некоторые накладные расходы и состязания.

Разбиение на основе диапазонов пропускает обычное перечисление на входной стороне и предварительно распределяет одинаковое количество элементов для каждого рабочего потока, избегая состязаний на входной последовательности. Но если случится так, что некоторые потоки получат простые элементы и завершатся раньше, то они окажутся в состоянии простоя, пока остальные потоки продолжат свою работу. Ранее приведенный пример с простыми числами может плохо выполняться при разбиении на основе диапазонов. Примером, когда такое разбиение оказывается удачным, является вычисление суммы квадратных корней первых 10 миллионов целых чисел:

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

Метод *ParallelEnumerable.Range* возвращает *ParallelQuery<T>*, поэтому вызывать *AsParallel* впоследствии не придется.

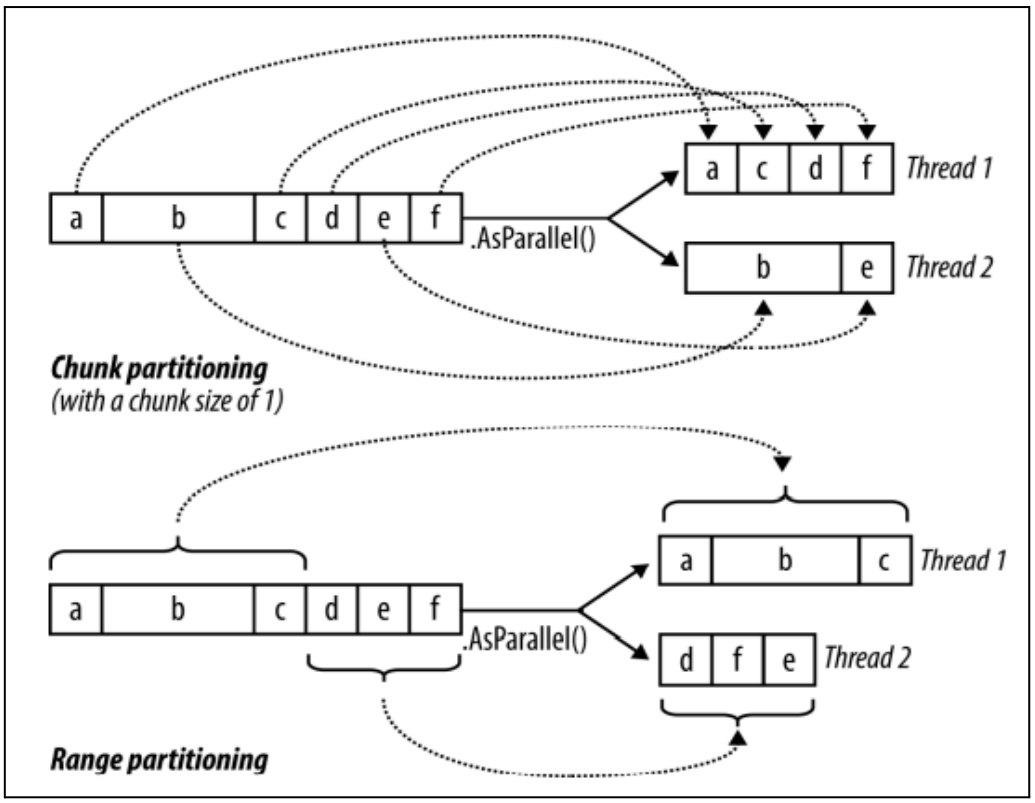

Рис. 3. Сравнение разбиения на основе порций и разбиения на основе диапазонов

- Разбиение на основе диапазонов необязательно распределяет диапазоны элементов в смежных блоках — взамен может быть выбрана “полосовая” стратегия. Например, при наличии двух рабочих потоков один из них может обрабатывать элементы в нечетных позициях, а другой — элементы в четных позициях. Операция *TakeWhile* почти наверняка инициирует полосовую стратегию, чтобы избежать излишней обработки элементов позже в последовательности.

Оптимизация специального агрегирования

Инфраструктура PLINQ эффективно распараллеливает операции *Sum*, *Average*, *Min* и *Max* без дополнительного вмешательства. Тем не менее операция *Aggregate* представляет особую трудность для PLINQ. Операция *Aggregate* выполняет специальное агрегирование. Например, следующий код суммирует последовательность чисел, имитируя операцию *Sum*:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate(0, (total, n) => total + n);    // 6
```

Для агрегаций без начального значения предоставляемый делегат должен быть ассоциативным и коммутативным. Если указанное правило нарушается, тогда инфраструктура PLINQ даст некорректные результаты, поскольку она извлекает множество начальных значений из входной последовательности, чтобы выполнять агрегирование нескольких частей последовательности одновременно.

Агрегации с явным начальным значением могут выглядеть как безопасный вариант для PLINQ, но, к сожалению, обычно они выполняются последовательно, т.к. полагаются на единственное начальное значение. Чтобы смягчить данную проблему, PLINQ предлагает еще одну перегруженную версию метода *Aggregate*, которая позволяет указывать множество начальных значений — или скорее функцию фабрики начальных значений. В каждом потоке эта функция выполняется для генерации отдельного начального значения, которое фактически становится локальным для потока накопителем, куда локально агрегируются элементы.

Потребуется также предоставить функцию для указания способа объединения локального и главного накопителей. Наконец, перегруженная версия метода Aggregate (отчасти беспричинно) ожидает делегат для проведения любой финальной трансформации результата (в принципе его легко обеспечить, просто выполняя нужную функцию на результате после его получения). Таким образом, ниже перечислены четыре делегата в порядке их передачи.

- *seedFactory*. Возвращает новый локальный накопитель.
- *updateAccumulatorFunc*. Агрегирует элемент в локальный накопитель.
- *combineAccumulatorFunc*. Объединяет локальный накопитель с главным накопителем.
- *resultSelector*. Применяет любую финальную трансформацию к конечному результату.

- ❖ В простых сценариях вместо фабрики начальных значений можно указывать просто величину начального значения. Такая тактика потерпит неудачу, когда начальное значение относится к ссылочному типу, который требуется изменять, потому что один и тот же экземпляр будет затем совместно использоваться всеми потоками.

В качестве очень простого примера ниже приведен запрос, который суммирует значения в массиве *numbers*:

```
new[] { 1, 2, 3 }.AsParallel().Aggregate(
  () => 0, // seedFactory
  (localTotal, n) => localTotal + n, // updateAccumulatorFunc
  // combineAccumulatorFunc
  (mainTot, localTot) => mainTot + localTot,
         finalResult => finalResult) // resultSelector
```

Пример несколько надуман, т.к. тот же самый результат можно было бы получить не менее эффективно с применением более простых подходов (скажем, с помощью агрегации без начального значения или, что еще лучше, посредством операции *Sum*). Чтобы предложить более реалистичный пример, предположим, что требуется вычислить частоту появления каждой буквы английского алфавита в заданной строке. Простое последовательное решение может выглядеть следующим образом:

```
string text = "Let's suppose this is a really long string";
var letterFrequencies = new int[26];
foreach (char c in text)
{
  int index = char.ToUpper(c) - 'A';
  if (index >= 0 && index <= 26) letterFrequencies[index]++;
};
```

- ❖ Примером, когда входной текст может оказаться очень длинным, являются генные цепочки. В таком случае "алфавит" состоит из букв *a*, *c*, *g* и *t*.

Для распараллеливания такого запроса мы могли бы заменить оператор *foreach* вызовом метода *Parallel.ForEach* (как будет показано в следующем разделе), но тогда пришлось бы иметь дело с проблемами параллелизма на разделяемом массиве.

Блокирование доступа к данному массиву решило бы проблемы, но уничтожило бы возможность распараллеливания.

Операция *Aggregate* предлагает более аккуратное решение. В этом случае накопителем выступает массив, похожий на массив *letterFrequencies* из предыдущего примера. Ниже представлена последовательная версия, использующая *Aggregate*:

```
int[] result =
  text.Aggregate(
    new int[26],                  // Создать "накопитель"
    (letterFrequencies, c) =>     // Агрегировать букву
                                  // в этот "накопитель"
    {
      int index = char.ToUpper(c) - 'A';
      if (index >= 0 && index <= 26) letterFrequencies[index]++;
      return letterFrequencies;
    });
```

А вот параллельная версия, в которой применяется специальная перегруженная версия *Aggregate* из PLINQ:

```
int[] result =
  text.AsParallel().Aggregate(
    () => new int[26],  // Создать новый локальный накопитель

    (localFrequencies, c) =>  // Агрегировать в этот локальный накопитель
    {
      int index = char.ToUpper(c) - 'A';
      if (index >= 0 && index < 26) localFrequencies[index]++;
      return localFrequencies;
    },
    // Агрегировать локальный и главный накопители
    (mainFreq, localFreq) =>
      mainFreq.Zip(localFreq, (f1, f2) => f1 + f2).ToArray(),

    finalResult => finalResult     // Выполнить финальную трансформацию
  );                               // конечного результата
```

Обратите внимание, что функция локального накопителя изменяет массив *localFrequencies*. Возможность выполнения такой оптимизации важна – и она законна, поскольку массив *localFrequencies* является локальным для каждого потока.