Технология PLINQ

- Parallel LINQ = Parallel Language Integrated Query
- Для большинства LINQ-операторов есть параллельные версии
- Вызов параллельных операторов осуществляется при изменении входной структуры данных с помощью метода AsParallel()

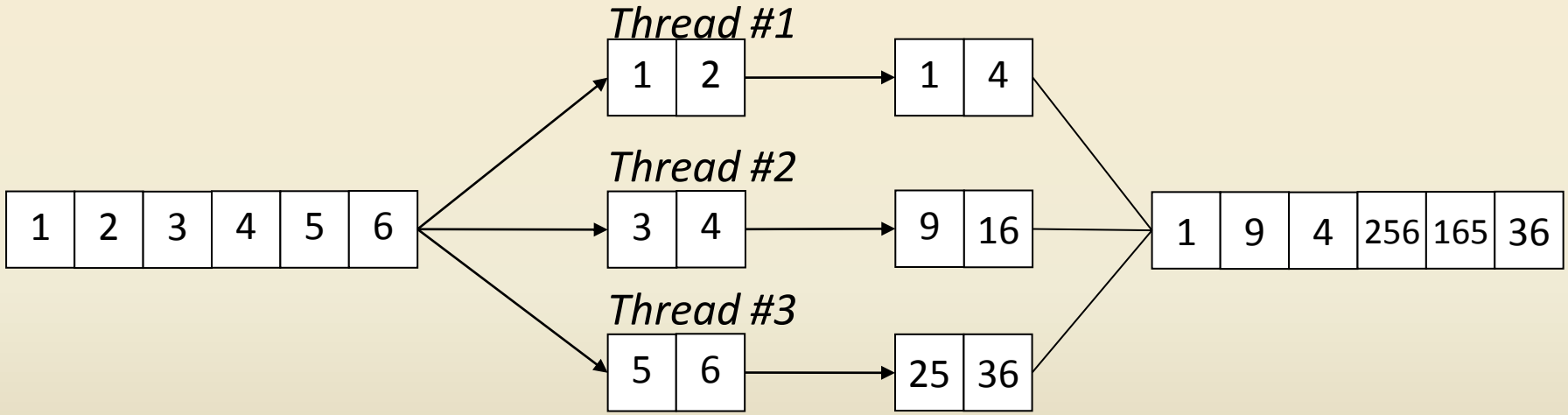
```
var q = data.AsParallel().Select(x => f(x));
```

PLINQ

```
// Входная последовательность элементов
var 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);
```

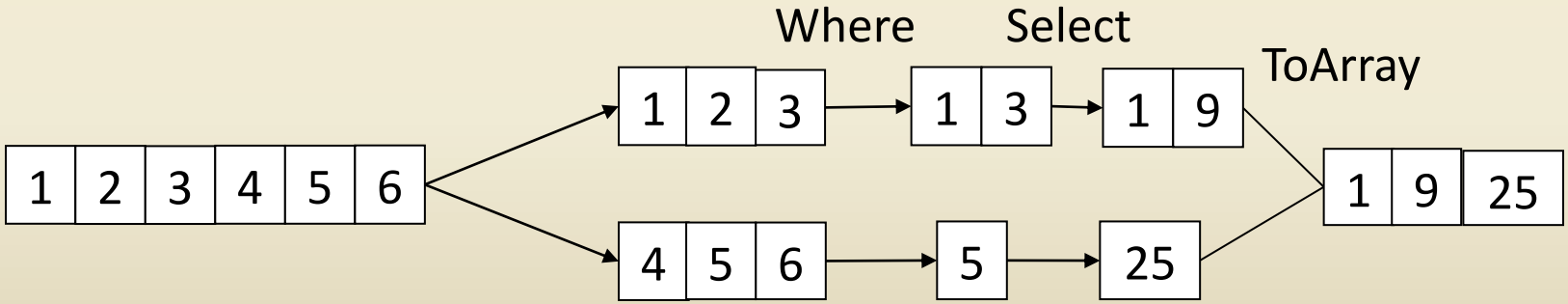
PLINQ: how to..

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



Преимущества PLINQ

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.



PLINQ vs. Parallel.ForEach

- PLINQ агрегирует результаты обработки в итоговую последовательность. В случае Parallel.ForEach необходимо использовать конкурентную коллекцию.
- PLINQ позволяет осуществлять несколько этапов обработки элементов при возможном изменении числа элементов и типа элементов.

Эффективность распараллеливания

- При распараллеливании LINQ-запросов появляются накладные расходы на разделение данных, агрегирование результатов
- Запросы Take, TakeWhile, Skip, SkipWhile работают только с исходным порядком элементов. Анализатор PLINQ не распараллеливает такие запросы или поддерживает порядок элементов.
- Не распараллеливаются *индексные перегрузки* методов Select, Where, если предыдущие операторы привели к изменению индексов элементов в структуре.

«Вынужденный» параллелизм

- Для параллельного выполнения запроса вне зависимости от целесообразности применяется специальный метод, который вызывается для параллельной структуры данных

```
var needToParallel = Enumerable.Range(0, 1000).AsParallel()
    .WithExecutionMode(
        ParallelExecutionMode. ForceParallelism)
    .Where(x =>
        {       Thread.SpinWait(1000000);  return true;       })
    .Select((x, i) => i).ToArray();
```

Степень параллелизма

- Существует возможность установить число потоков для обработки PLINQ-запроса
- Метод WithDegreeOfParallelism с указанием числа потоков вызывается для параллельной структуры данных

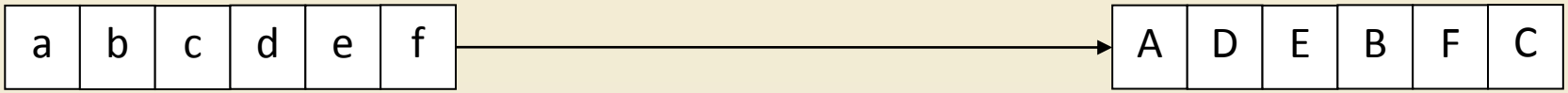
Буферизация

- PLINQ поддерживает три режима буферизации вычисления результатов
- По умолчанию используется авто-буферизация (AutoBuffered) - объем буфера для вычисления результатов определяется исполняющей средой
- Полная буферизация (fully-buffered) позволяет выполнить запрос полностью до предоставления результатов вне зависимости от числа элементов
- Третий режим не использует буферизацию (NotBuffered) – каждый элемент вычисляется по мере необходимости.
- Установка режима буферизации выполняется с помощью метода WithMergeOptions

Порядок элементов

При выполнении PLINQ-запроса порядок элементов в выходной последовательности по умолчанию не определен.

```
select Char.ToUpper(c)
```



Для сохранения порядка элементов применяются модификаторы AsOrdered(), AsUnordered()

AsOrdered

```
var query =
    (from numbers in source.AsParallel().AsOrdered()
    where numbers % 5 == 0
     select numbers).Take(10);
```

Разделение данных

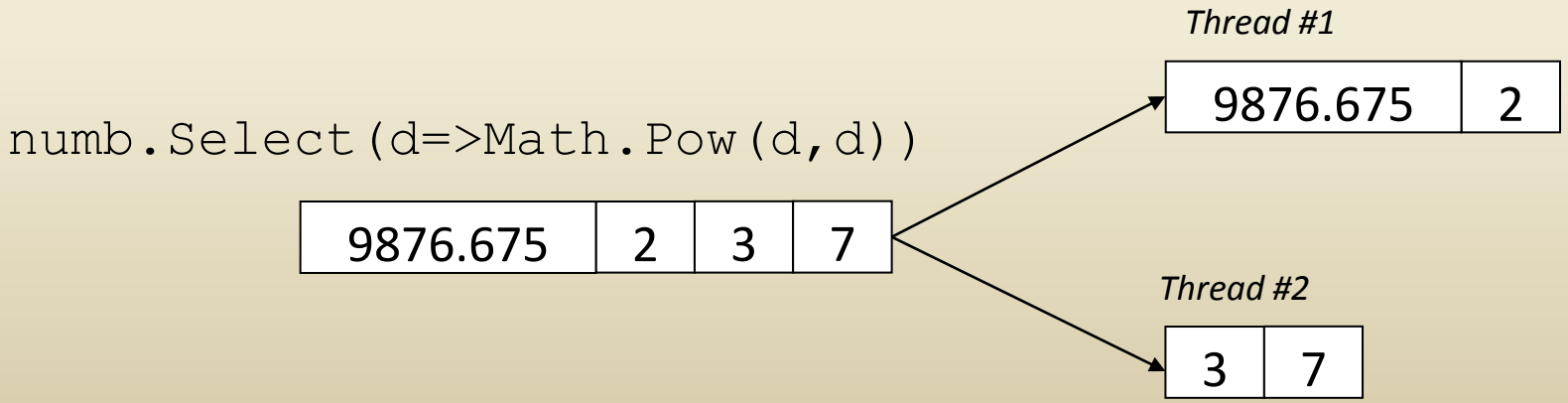
- Разделение по диапазону (Range partitioning) или статическая декомпозиция
- Хэш-секционирование (Hash partitioning)
- Блочная (динамическая) декомпозиция (chunk-partitioning)

Декомпозиция. Хэш-секционирование

- Хэш-секционирование выполняется для операторов, сравнивающих элементы: GroupBy, Join, GroupJoin, Intersect, Except, Union, Distinct.
- Хэш-секционирование требует расчета хэш-значений для всех элементов последовательности; элементы с одинаковыми хэш-значениями обрабатываются одним и тем же потоком.
- Декомпозиция по хэш-значениям выбирается планировщиком при необходимости и не может быть установлена пользователем
- Такая декомпозиция является наиболее медленной из-за необходимости вычисления хэш-значений.

Декомпозиция. Разделение по диапазону

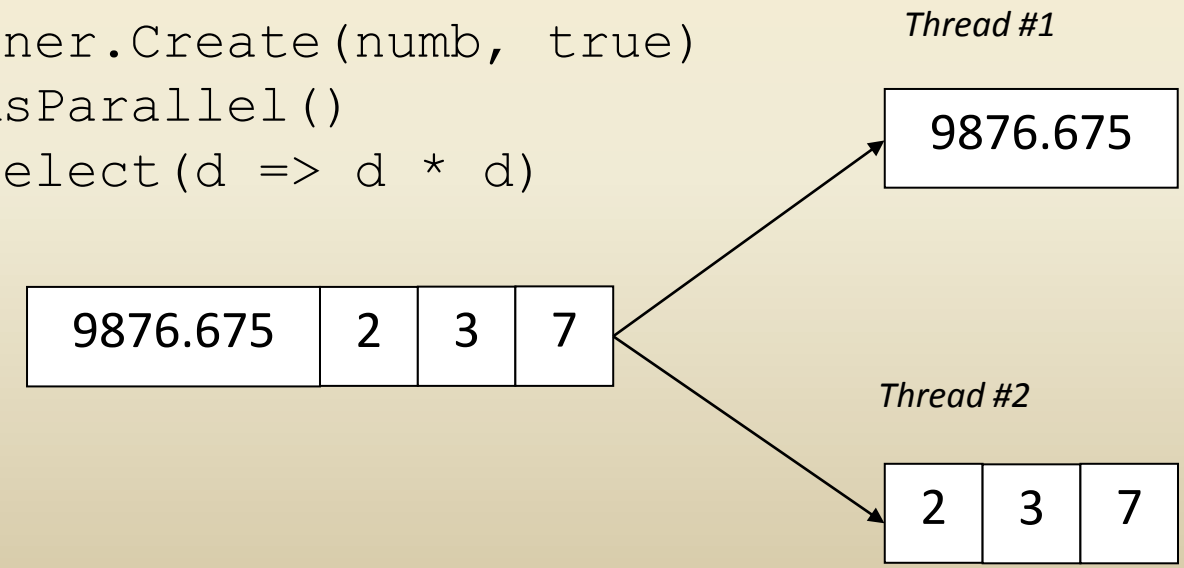
- При разделении по диапазону последовательность разбивается на равное число элементов, каждая порция обрабатывается в одном рабочем потоке.
- Такое разделение является достаточно эффективным, так как приводит к полной независимости обработки элементов на разных потоках и не требует какой-либо синхронизации.



Декомпозиция. Блочная декомпозиция

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.

```
Partitioner.Create(numb, true)
        .AsParallel()
        .Select(d => d * d)
```



Агрегированные вычисления

```
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))
);
```