



ОНЛАЙН-ОБРАЗОВАНИЕ


Онлайн-образование

Проверить, идет ли запись!





Меня хорошо видно && слышно?

Ставьте  , если все хорошо
Напишите в чат, если есть проблемы

Внутрипроцессное взаимодействие



Виктор Дзицкий

TeamLead, Full Stack .Net Developer

SolarLab

Telegram: @Dzitskiy

Правила вебинара



Активно участвуем



Задаем вопрос в чат или голосом



Off-topic обсуждаем в Slack #канал группы или #general



Вопросы вижу в чате, могу ответить не сразу



Тестирование

План вебинара

Параллельное
программирование



Parallel LINQ



Parallel library



Concurrent collections

Цели вебинара

1

Применять потоки, задачи, Parallel LINQ

2

Распараллеливать расчёты для ускорения вычислений

Смысл вебинара

1

Максимально эффективно использовать ресурсы ЦП и памяти для улучшения производительности

2

Параллельные коллекции: обработка сценариев с высокой пропускной способностью

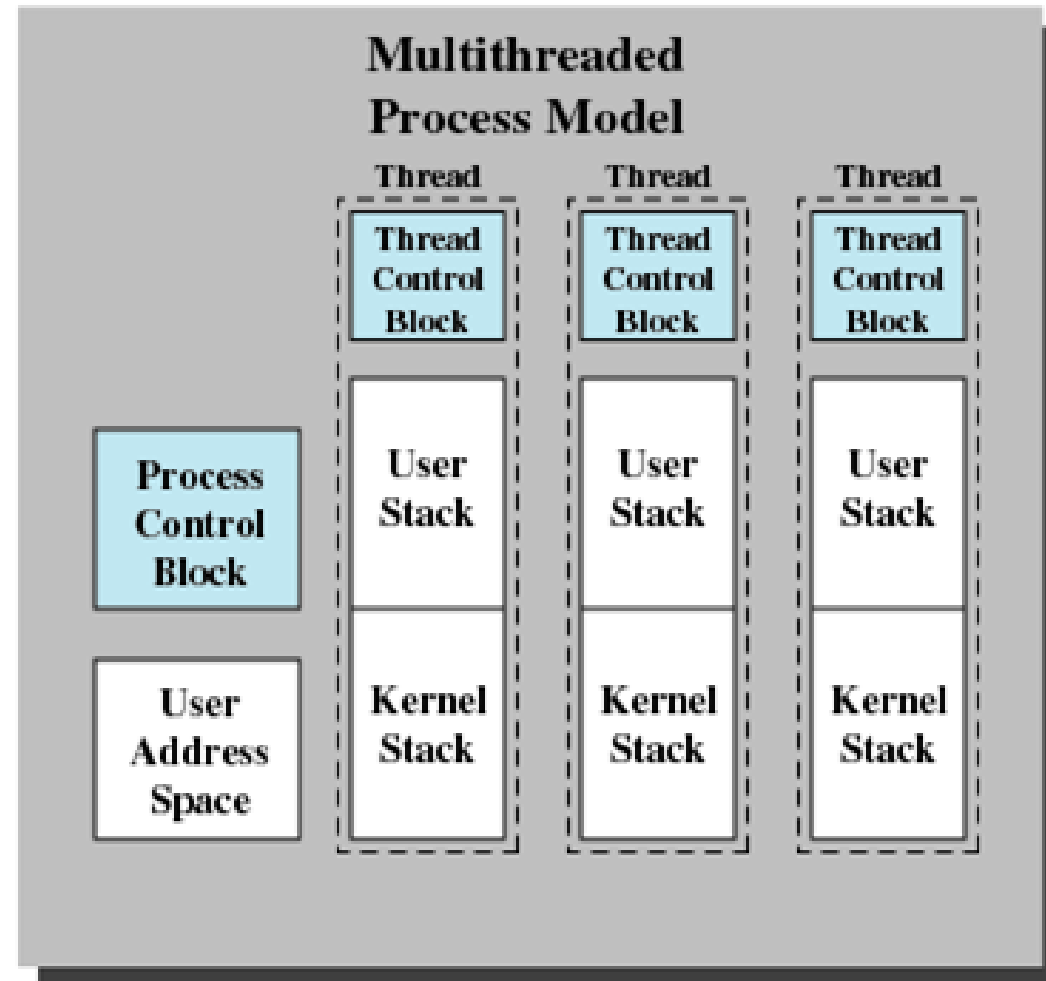
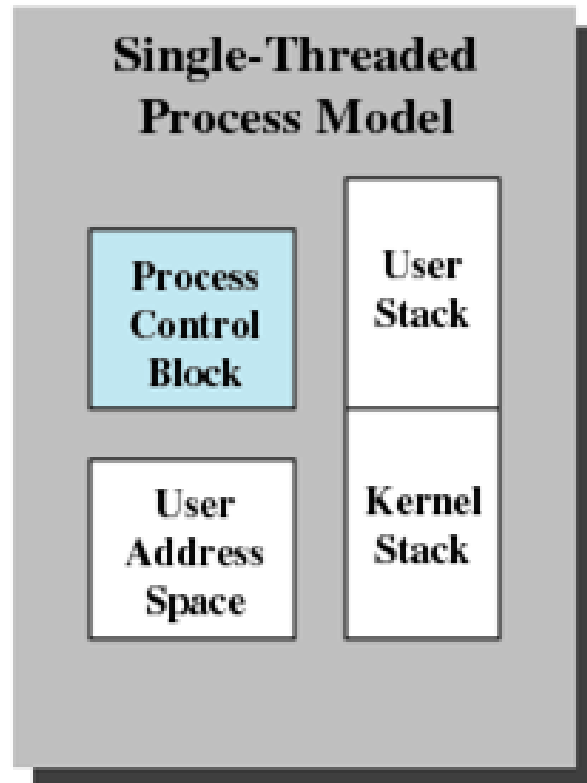
3

Обработка в реальном времени

The background of the slide is a high-angle, blue-tinted aerial photograph of a city skyline, likely New York City, showing numerous skyscrapers and buildings. Overlaid on this image is a semi-transparent blue band that contains a white network diagram. This diagram consists of numerous small dots (nodes) connected by thin white lines, creating a complex web-like structure that spans the width of the slide. Centered within this blue band is the main title text.

Многопоточность и параллелизм

Однопоточная и многопоточная модель процесса



Многопоточность

Использовать несколько ядер легко для большинства серверных приложений, где каждый поток может независимо обрабатывать отдельный клиентский запрос, но сложнее на настольных компьютерах, поскольку обычно требуется, чтобы вы взяли код с интенсивными вычислениями и сделали следующее:

1. *Разделите его на небольшие куски.*
2. *Выполняйте эти фрагменты параллельно с помощью многопоточности.*
3. *Сопоставляйте результаты по мере их поступления, потокобезопасным и производительным способом.*

Параллелизм

Существует две стратегии разделения работы между потоками:

1. Параллелизм данных
 - лучше масштабируется
 - уменьшает конфликты
 - структурированный
 - менее подвержен ошибкам
2. Параллелизм задач

Модели параллелизма в программах

Императивный параллелизм задач. Явное выражение операторов программы, выполнение которых должно быть выполнено параллельно.

Пример: *Явное использование потоков в программировании.*

Императивный параллелизм данных. Необходимо выразить императивный оператор, ориентированный на обработку данных. Параллелизм выполнения оператора – задача среды разработки.

Пример: *операторы `for`, `foreach` (в PLINQ).*

Декларативный параллелизм данных. Необходимо выразить желаемый результат обработки данных. Параллелизм выполнения – задача среды разработки.

Пример: *SQL, PLINQ.*



Императивный параллелизм задач

Преимущества использования потоков

- Создание потока занимает меньше времени, чем создание процесса.
- Поток можно завершить быстрее, чем процесс.
- Переключение между потоками процесса происходит намного быстрее.
- Потоки одного процесса разделяют его ресурсы, что позволяет потокам взаимодействовать без привлечения ядра ОС.

Потоки на однопроцессорной системе

- Работа в приоритетном (foreground) и фоновых режимах (background).
 - Электронные таблицы
- Асинхронная обработка
 - Работа с низкопроизводительными устройствами
- Повышение скорости выполнения
 - Одновременные операции по загрузке и обработке данных
- Модульная структура программы
 - Осуществление разнообразных действий и/или выполняющие множество операций ввода-вывода из и на различные источники.

Основные состояния потока

- События, изменяющие состояние потока

1. Порождение (Spawn)

- Первичный поток процесса порождается ОС. Другие потоки процесса порождаются потоками (начиная с первичного).
- Новый поток создаётся со своим контекстом и стеками и помещается в очередь готовых к выполнению.

2. Блокирование

- Ожидание некоторого события с сохранением контекста
- Процессор переходит к выполнению другого потока.

3. Разблокирование

- Наступление события и перевод потока в готовое состояние

4. Завершение

- Разрушение контекста и стеков потока.

- Приостановка (Suspending) процесса приводит к простановке всех его потоков

- Завершение процесса приводит к завершению всех его потоков.

Вызов WinApi функций

```
#include <iostream>
```

```
#include <windows.h>
```

```
// Функция, исполняемая потоками
DWORD __stdcall CharsThread(LPVOID p)
{
    // LPVOID p есть указатель на символ
    TCHAR c = *(TCHAR*)(p);
    // Бесконечно выводим на экран символ
    while(1)
        std::cout << c;
    return 0;
}
```

```
int _tmain(int argc, _TCHAR* argv[]) {
    TCHAR asterisk = '*';
    HANDLE hThread = CreateThread(NULL, 0, CharsThread, &asterisk, 0, NULL);

    // Время ожидания завершения потоков
    #define TIMEOUT 2000
    // Ждём завершения потока TIMEOUT миллисекунд
    WaitForSingleObject(hThread, TRUE, TIMEOUT);
    /* Закрываем дескриптор потока. Сам поток продолжает работать до завершения процесса */
    CloseHandle(hThread);
    return 0;
}
```

```
[DllImport("Kernel32.dll", CharSet =
    CharSet.Auto, SetLastError = true)]
private unsafe static extern uint CreateThread(
    uint* lpThreadAttributes,
    uint dwStackSize,
    ThreadStart lpStartAddress,
    uint* lpParameter,
    uint dwCreationFlags,
    out uint lpThreadId);
```


Потоки в С#

```
Thread t = Thread.CurrentThread; // получаем текущий поток
Console.WriteLine($"Имя потока: {t.Name}"); //получаем имя потока, состояние и др.
Console.WriteLine($"Статус потока: {t.ThreadState}");
Console.WriteLine($"Домен приложения: {Thread.GetDomain().FriendlyName}");
```

```
Thread thread = new Thread(new ThreadStart(метод)); // создаем новый поток
thread.Start(); // запускаем поток // создаем новый поток
Thread thread = new Thread(new ParameterizedThreadStart(метод, принимающий object));
thread.Start(параметр);
```

```
TimerCallback tm = new TimerCallback(метод); // устанавливаем метод обратного вызова
Timer timer = new Timer(tm, num, 0, 2000); // создаем таймер
```

Задачи в C#

```
Task task = new Task(() => Console.WriteLine("Hello Task!"));  
task.Start();  
task.Wait();
```

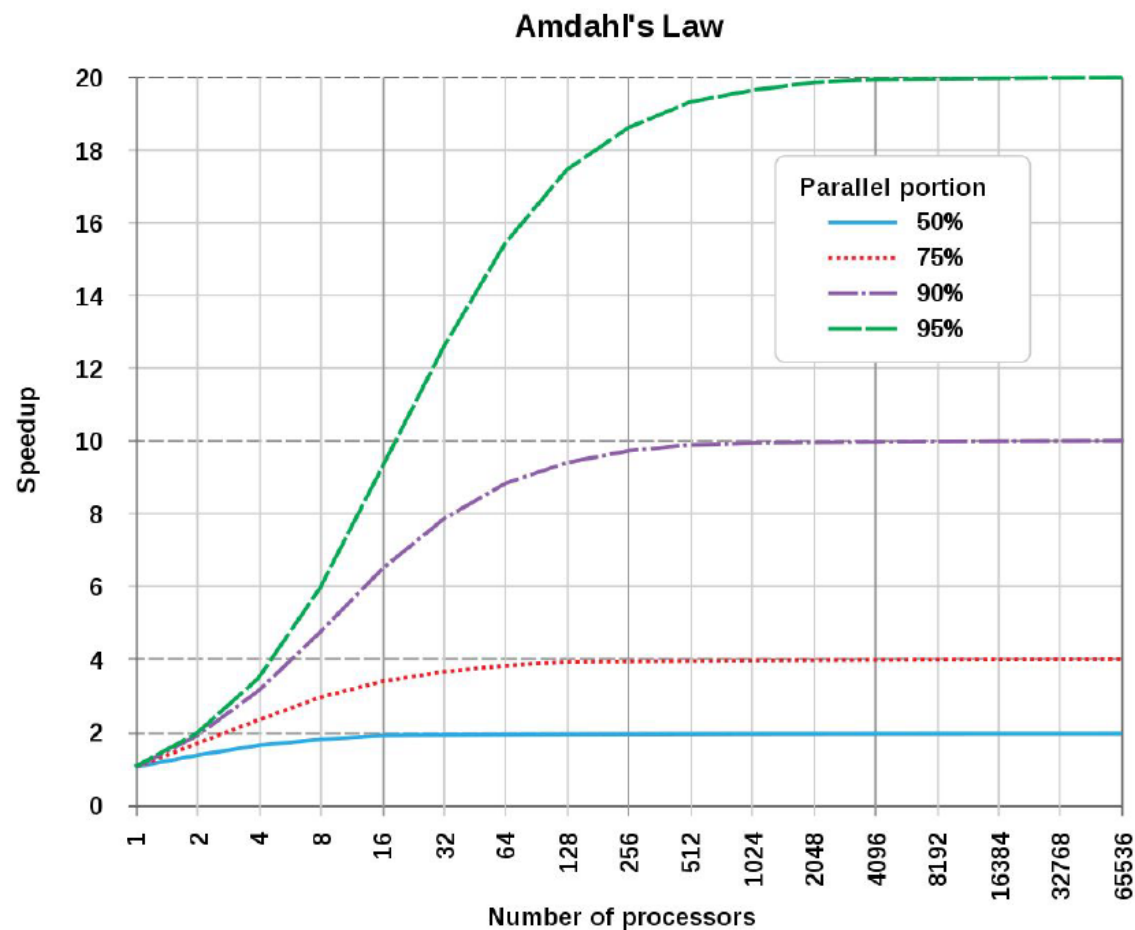
```
Task task = Task.Run(() => Console.WriteLine("Hello Task!"));
```

```
Task task = Task.Factory.StartNew(() => Console.WriteLine("Hello Task!"));
```


The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue and green gradient. A network of thin, light blue lines connects various points across the image, creating a digital or technological feel. The word "ДЕМО" is prominently displayed in the center in a bold, white, sans-serif font.

ДЕМО

Закон Амдала



Проблемой в использовании многоядерности является закон Амдала, который гласит, что максимальное улучшение производительности за счет распараллеливания определяется частью кода, которая должна выполняться последовательно. Например, если только две трети времени выполнения алгоритма можно распараллелить, вы никогда не сможете превысить трехкратный прирост производительности — даже при бесконечном количестве ядер.



Декларативный параллелизм задач

Задачи в C#

`Parallel.Invoke(массив делегатов);`

`Parallel.For(1, 10, метод обработки);`

`Parallel.ForEach<int>(new List<int>() { 1, 3, 5, 8 },
метод обработки);`

The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue and green gradient. A network of thin, light blue lines connects various points across the gradient, creating a digital or technological feel. The word "ДЕМО" is centered in the middle of the image in a large, white, sans-serif font.

ДЕМО



Декларативный параллелизм данных

Parallel LINQ

```
static void Main(string[] args)
{
    int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, };
    var results = from n in numbers.AsParallel()
                  select МетодВычисления(n);
    foreach (var n in results)
        Console.WriteLine(n);
    Console.ReadLine();
}
```

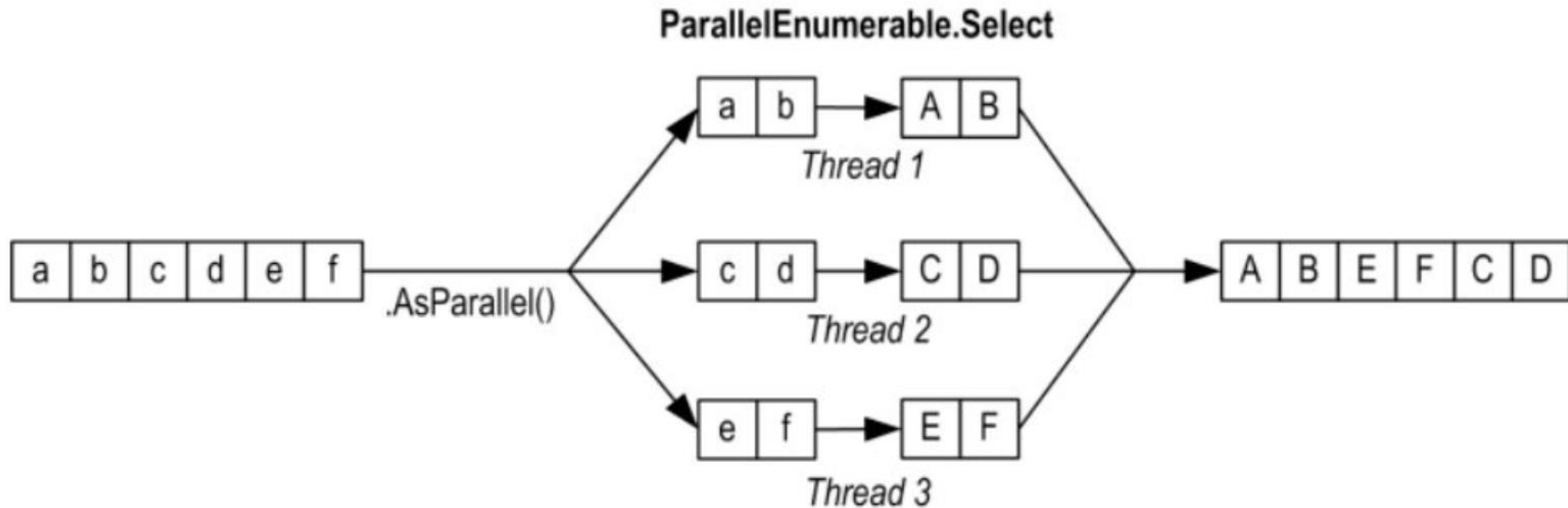
.AsOrdered()

.AsUnordered()

.ForAll()

.WithCancellation(new CancellationTokenSource().Token)

Parallel LINQ



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


Методы Parallel LINQ

AsParallel	Точка входа для PLINQ. Указывает, что по возможности остальная часть запроса должна быть параллелизована.
AsSequential	Указывает, что остальная часть запроса должна выполняться последовательно, как непараллельный запрос LINQ.
AsOrdered	Указывает, что PLINQ должен сохранить порядок исходной последовательности до конца запроса либо до тех пор, пока порядок не изменится
AsUnordered	Указывает, что PLINQ для остальной части запроса не обязан сохранять порядок исходной последовательности.

The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue filter. A prominent network pattern, consisting of interconnected white dots and lines, is visible across the center of the image, creating a digital or technological aesthetic. The word "ДЕМО" is centered in the middle of the image in a large, white, sans-serif font.

ДЕМО

AsSequential

AsSequential() необходимо перед вызовом методов, которые имеют побочные эффекты или не являются потокобезопасными.

Для операторов запроса, которые принимают две входные последовательности (**Join**, **GroupJoin**, **Concat**, **Union**, **Intersect**, **Except** и **Zip**), необходимо применить **AsParallel()** к обеим входным последовательностям (в противном случае выдается исключение).

```
mySequence.AsParallel()  
            .Where (n => n > 100)  
            .AsParallel()  
            .Select (n => n * n) |  
// Wraps sequence in ParallelQuery<int>  
// Outputs another ParallelQuery<int>  
// Unnecessary - and inefficient!
```

AsOrdered, AsUnordered

Вызов **AsOrdered** приводит к снижению производительности при большом количестве элементов, поскольку PLINQ должен отслеживать исходное положение каждого элемента.

Вы можете свести на нет эффект **AsOrdered** последующего выполнения запроса, вызвав **AsUnordered**: это вводит «случайную точку перемешивания», которая позволяет запросу выполняться более эффективно с этого момента.

```
inputSequence.AsParallel().AsOrdered()  
    .QueryOperator1()  
    .QueryOperator2()  
    .AsUnordered()           // From here on, ordering doesn't matter  
    .QueryOperator3()
```


Ограничения Parallel LINQ

1. PLINQ не всегда работает параллельно:

Следующие операторы запроса предотвращают распараллеливание запроса, если только исходные элементы не находятся в исходной индексной позиции:

Take, TakeWhile, Skip, и SkipWhile

Следующие операторы запроса можно распараллелить, но они используют дорогостоящую стратегию секционирования, которая иногда может быть медленнее, чем последовательная обработка:

Join, GroupBy, GroupJoin, Distinct, Union, Intersect и Except

- 2. PLINQ предназначен только для локальных коллекций: он не работает с LINQ to SQL или Entity Framework
- 3. Если запрос PLINQ выдает исключение, оно выдается повторно как свойство, **AggregateException** чье **InnerExceptions** свойство содержит реальное исключение (или исключения).

Дополнительные методы Parallel LINQ

WithMergeOptions	Предоставляет подсказку о том, каким образом PLINQ должен объединять параллельные результаты в одну последовательность в потоке-потребителе, если это возможно.
WithDegreeOfParallelism	Указывает максимальное количество процессоров, которое PLINQ должен использовать для параллелизации запроса.
WithCancellation	Указывает, что PLINQ должен периодически отслеживать состояние предоставленного токена отмены и отменить выполнение, если он будет запрошен.
WithExecutionMode	Указывает, должен ли PLINQ параллелизовать запрос, даже если по умолчанию он должен выполняться последовательно.

WithMergeOptions

Вы можете настроить поведение буферизации PLINQ, вызвав `WithMergeOptions` после `AsParallel`.

- `AutoBuffered` - Значение по умолчанию обычно дает наилучшие общие результаты.
- `NotBuffered` - отключает буфер и полезно, если вы хотите увидеть результаты как можно скорее;
- `FullyBuffered` - кэширует весь набор результатов перед представлением его потребителю (операторы `OrderBy` и `Reverse`, естественно, работают таким же образом, как и операторы элемента, агрегации и преобразования).

Функциональная чистота

выполнять потокобезопасные операции.

```
int i = 0;  
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

Для достижения наилучшей производительности любые методы, вызываемые из операторов запроса, должны быть потокобезопасными, поскольку они не записывают данные в поля или свойства (без побочных эффектов или *функционально чистые*). Если они потокобезопасны благодаря блокировке, потенциал параллелизма запроса будет ограничен — длительностью блокировки, разделенной на общее время, затраченное в этой функции.

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


Вызов функций с интенсивным вводом выводом

```
from site in new[]
{
    "www.albahari.com",
    "www.linqpad.net",
    "www.oreilly.com",
    "www.takeonit.com",
    "stackoverflow.com",
    "www.rebeccarey.com"
}
.AsParallel().WithDegreeOfParallelism(6)
let p = new Ping().Send (site)
select new
{
    site,
    Result = p.Status,
    Time = p.RoundtripTime
}
```

PLINQ может эффективно распараллеливать такие запросы, вызывая **WithDegreeOfParallelism** после AsParallel.

Отмена операций

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

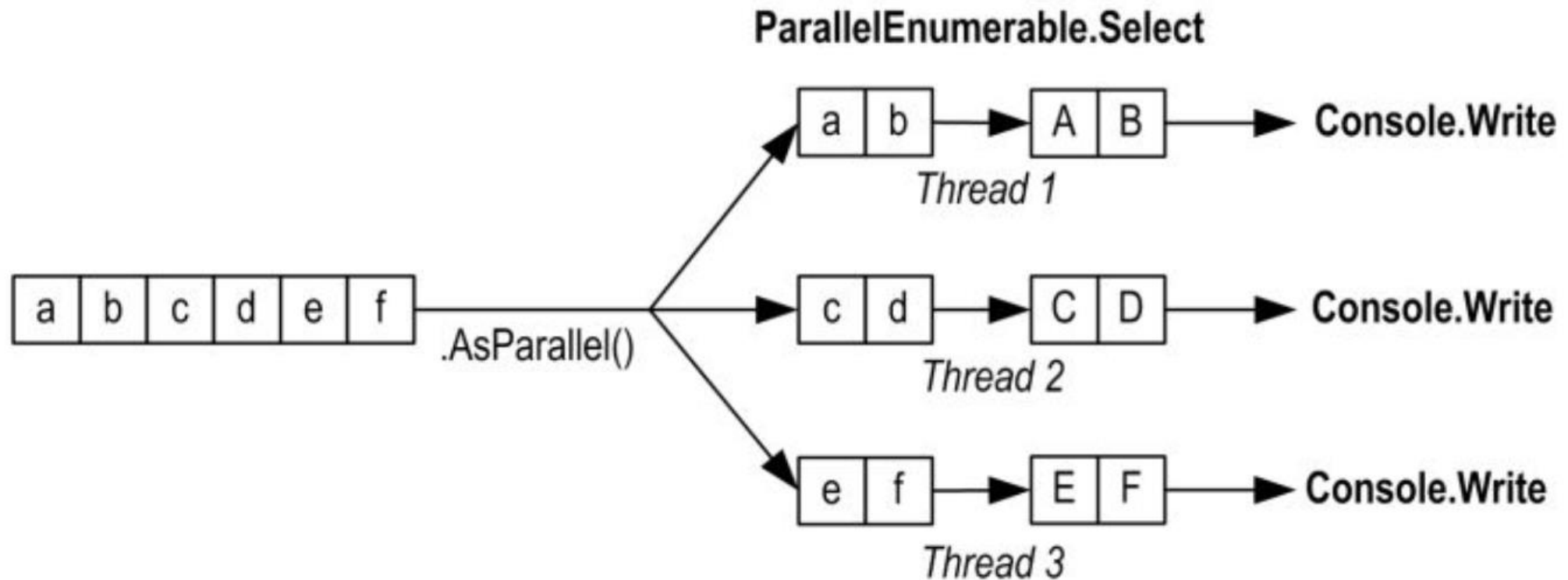
var cancelSource = new CancellationTokenSource();
var primeNumberQuery =
    from n in million.AsParallel().WithCancellation (cancelSource.Token)
    where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
    select n;

new Thread (() => {
    Thread.Sleep (100);    // Cancel query after
                           // 100 milliseconds.
    cancelSource.Cancel();
}).Start();

try
{
    // Start query running:
    int[] primes = primeNumberQuery.ToArray();
    // We'll never get here because the other thread will cancel us.
}
catch (OperationCanceledException)
{
    Console.WriteLine ("Query canceled");
}
```

Чтобы вставить токен, вызовите WithCancellation после вызова AsParallel, передав Token свойство объекта CancellationTokenSource. Затем другой поток может вызвать Cancel, который выдает OperationCanceledException

Оптимизация на стороне вывода

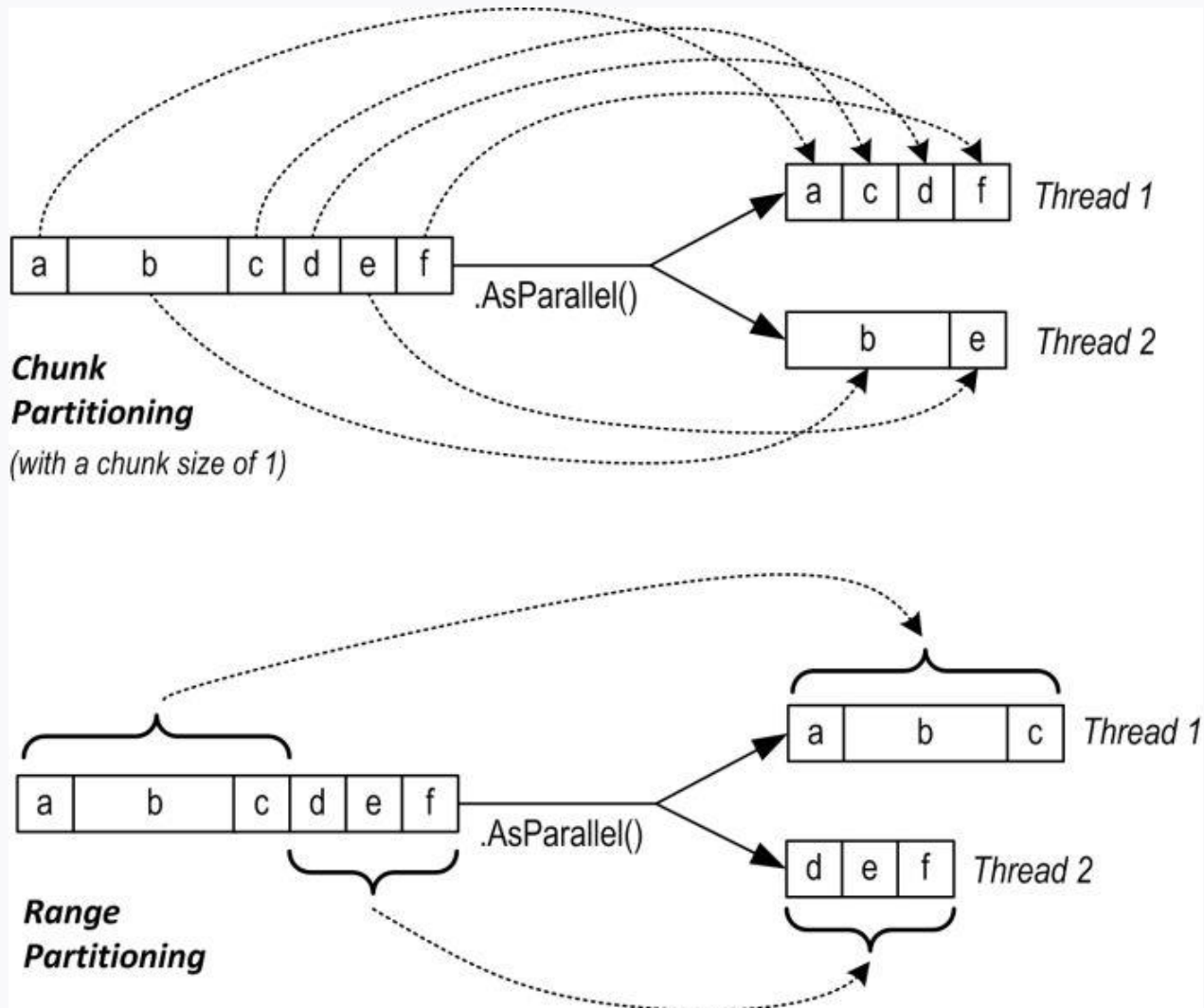


```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write)
```


The image features a high-angle, aerial view of a dense urban skyline, likely New York City, with numerous skyscrapers and buildings. The entire image is overlaid with a semi-transparent blue filter. A prominent network pattern, consisting of interconnected white lines and dots, is visible across the center of the image, creating a digital or technological aesthetic. The word "ДЕМО" is centered in the middle of the image in a large, white, sans-serif font.

ДЕМО

Оптимизация на стороне ввода



3 стратегии:

1. хэш-секционирование
2. секционирование по диапазону
3. фрагментирование

Partitioner

1. Балансировка нагрузки. Разделяя коллекцию на более мелкие фрагменты, Partitioner может помочь более равномерно сбалансировать нагрузку между потоками, предотвращая сценарии, в которых некоторые потоки завершаются намного раньше, чем другие.
2. Эффективное использование ресурсов: разделы могут обрабатываться независимо, что позволяет лучше использовать ресурсы ЦП.
3. позволяет использовать индивидуальные стратегии разделения данных на основе конкретных требований, что может быть более эффективным, чем методы секционирования по умолчанию.

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };  
var parallelQuery =  
    Partitioner.Create (numbers, true).AsParallel()  
    .Where (...)
```


The background of the slide is a high-angle, blue-tinted aerial photograph of a dense urban skyline, likely New York City. Overlaid on this image is a semi-transparent network diagram consisting of numerous small blue dots connected by thin, light-blue lines, creating a web-like pattern across the center of the slide. The title text is centered within this network area.

Concurrent Collections

Классы параллельных коллекций

System.Collections.Concurrent

1. BlockingCollection<T>
2. ConcurrentBag<T>
3. ConcurrentDictionary<TKey,TValue>
4. ConcurrentQueue<T>
5. ConcurrentStack<T>

System.Collections.Concurrent

Параллельные коллекции иногда могут быть полезны в общей многопоточности, когда вам нужна потокобезопасная коллекция. Однако есть некоторые предостережения:

- Параллельные коллекции настроены для *параллельного программирования*. Обычные коллекции превосходят их во всех сценариях, кроме сценариев с высокой степенью параллелизма.
- Потокобезопасная коллекция не гарантирует, что [использующий ее код будет потокобезопасным](#).
- Если вы перебираете параллельную коллекцию, пока другой поток ее изменяет, исключение не создается. Вместо этого вы получаете смесь старого и нового контента.
- Не существует параллельной версии **List<T>**.
- Классы параллельного стека, очереди и Bag реализуются внутри с помощью связанных списков. Это делает их менее эффективными с точки зрения использования памяти, чем непараллельные классы **Stack** и **Queue** классы, но лучше подходят для одновременного доступа, поскольку связанные списки способствуют реализациям без блокировок или с низким уровнем блокировок. (Это связано с тем, что вставка узла в связанный список требует обновления всего пары ссылок, а вставка элемента в **List<T>**-подобную структуру может потребовать перемещения тысяч существующих элементов.)

IProducerConsumerCollection<T>

Коллекция производитель/потребитель — это коллекция, для которой есть два основных варианта использования:

- Добавление элемента («производство»)
- Получение элемента при его удалении («потреблении»)

Классическими примерами являются стеки и очереди. Коллекции производитель/потребитель играют важную роль в параллельном программировании, поскольку они способствуют эффективной реализации без блокировок.

Интерфейс `IProducerConsumerCollection<T>` представляет собой потокобезопасную коллекцию производитель/потребитель. Следующие классы реализуют этот интерфейс:

- `ConcurrentStack < T >`
- `ConcurrentQueue < T >`
- `ConcurrentBag < T >`

System.Collections.Concurrent

Блокирующая коллекция оборачивает любую коллекцию, реализующую [IProducerConsumerCollection<T>](#) , и позволяет использовать **Take** элемент из обернутой коллекции — блокировка, если элемент недоступен.

Блокирующая коллекция также позволяет ограничить общий размер коллекции, блокируя *производителя* , если этот размер превышен. Коллекция, ограниченная таким образом, называется *ограниченной блокирующей коллекцией* .

Использовать **BlockingCollection<T>**:

1. Создайте экземпляр класса, при необходимости указав [IProducerConsumerCollection<T>](#) для переноса и максимальный размер (привязку) коллекции.
2. Вызовите **Add**или , **TryAdd** чтобы добавить элементы в базовую коллекцию.

ConcurrentBag<T>

ConcurrentBag<T> хранит неупорядоченную коллекцию объектов (с возможностью дублирования).

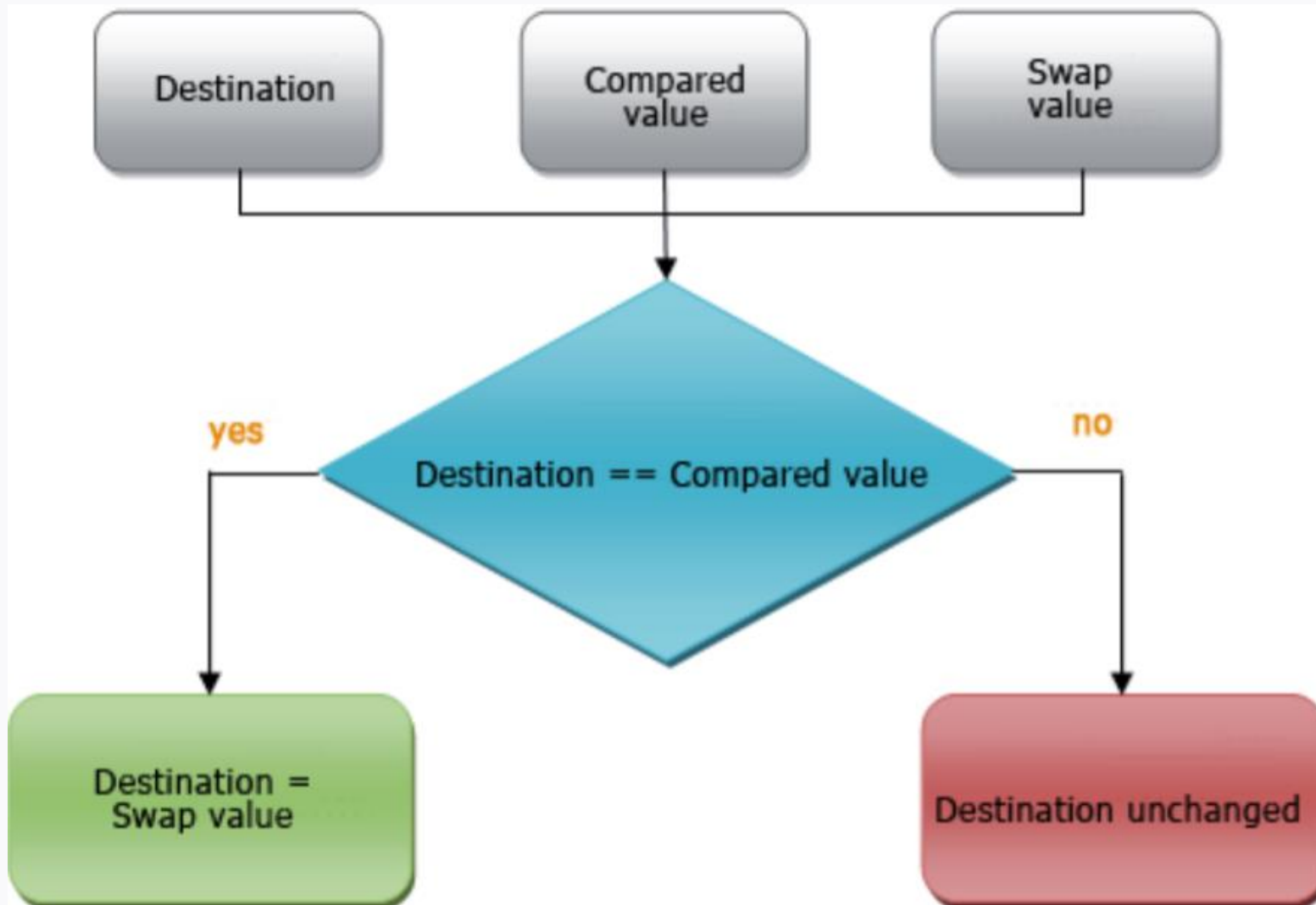
ConcurrentBag<T> подходит в ситуациях, когда вам *не важно*, какой элемент вы получите при вызове **Take** или **TryTake**.

Преимущество **ConcurrentBag<T>** параллельной очереди или стека заключается в том, что метод пакета **Add** практически *не* подвергается конфликтам при вызове многими потоками одновременно. Напротив, **Add** параллельный вызов очереди или стека вызывает *некоторые* конфликты (хотя и гораздо меньшие, чем блокировка *непараллельной* коллекции). Вызов **Take** параллельного пакета также очень эффективен — до тех пор, пока каждый поток не принимает больше элементов, чем ему **Add** нужно.

Внутри параллельного пакета каждый поток получает собственный частный связанный список. Элементы добавляются в частный список, принадлежащий вызывающему потоку **Add**, что устраняет конфликты. Когда вы перебираете пакет, перечислитель проходит через частный список каждого потока, по очереди получая каждый из его элементов.

Когда вы вызываете **Take**, пакет сначала просматривает личный список текущего потока. Если есть хотя бы один элемент, он сможет выполнить задачу легко и (в большинстве случаев) без конфликтов. Но если список пуст, ему придется «украсть» элемент из частного списка другого потока, что приведет к потенциальному конфликту.

CAS



Interlocked.CompareExchange

```
4 public void Push(T value)
5 {
6     var newNode = new Node<int> { Data = value, Next = _head };
7
8     if (Interlocked.CompareExchange(ref _head, newNode, newNode.Next) != newNode.Next)
9     {
10         var spinner = new SpinWait();
11         do
12         {
13             spinner.SpinOnce();
14             newNode.Next = _head;
15         }
16         while (Interlocked.CompareExchange(ref _head, newNode, newNode.Next) != newNode.Next);
17     }
18 }
```


SpinLock и SpinWait

Их основное применение — написание пользовательских конструкций синхронизации.

SpinLock и **SpinWait** являются структурами, а не классами! Это дизайнерское решение представляло собой экстремальный метод оптимизации, позволяющий избежать затрат на косвенность и сборку мусора. Это означает, что вы должны быть осторожны, чтобы не *скопировать* экземпляры непреднамеренно — например, передав их другому методу без **ref** модификатора или объявив их как **readonly** поля. Это особенно важно в случае **SpinLock**.

Вопросы




Когда целесообразно применять параллельное программирование?



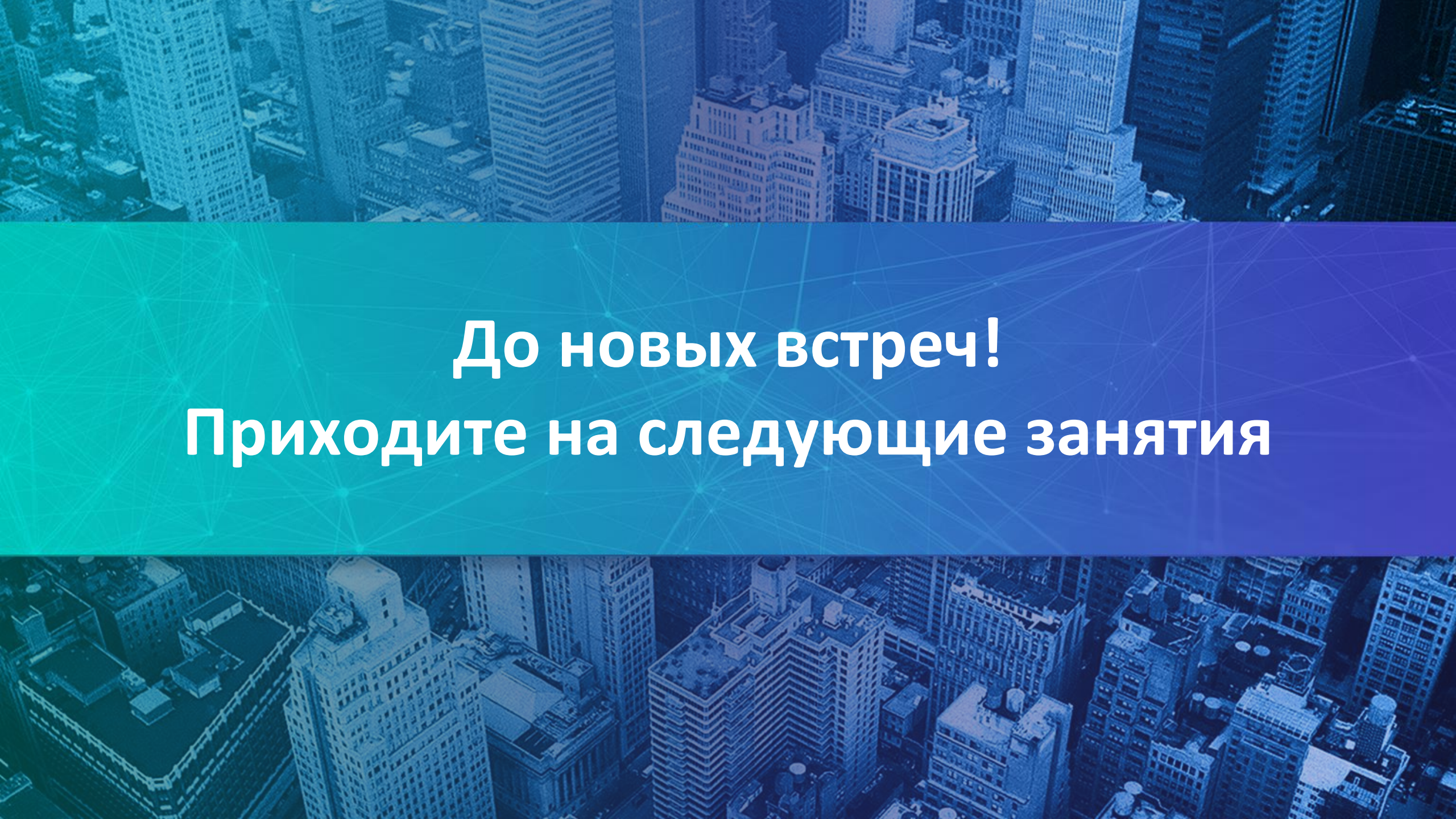
Чем отличаются императивный и декларативный подходы к распараллеливанию?



А в чем разница между параллелизмом задач и данных?

The background of the entire image is an aerial photograph of a city with many skyscrapers, likely New York City. The image is overlaid with a semi-transparent blue layer. In the center of this blue layer, there is a network of white lines connecting various points, creating a geometric pattern. The text is written in white, bold, sans-serif font, centered horizontally and vertically within the blue area.

Заполните, пожалуйста,
опрос о занятии по ссылке в чате



До новых встреч!
Приходите на следующие занятия