1. **TPL (Task Parallel Library):**

- TPL - это библиотека в .NET Framework, предназначенная для упрощения параллельного программирования. Одним из основных типов в TPL является `Task`. `Task` представляет собой абстракцию задачи, которая может быть выполнена асинхронно.

2. **Распараллеливание и количество элементов:**

- Эффект от распараллеливания наблюдается на большом количестве элементов из-за накладных расходов на создание и управление параллельными задачами. При малом количестве элементов эти накладные расходы могут превысить выигрыш от параллельного выполнения.

3. **Достоинства работы с задачами по сравнению с потоками:**

- Абстракция `Task` предоставляет удобный интерфейс для асинхронного программирования, автоматическую управляемость пулом потоков, поддержку отмены и обработку исключений.

4. **Способы создания и/или запуска Task:**

- `Task.Run(Action)`, `new Task(Action)`, `Task.Factory.StartNew(Action)`

5. **Методы Wait(), WaitAll() и WaitAny():**

- `Wait()` - блокирует вызывающий поток до завершения задачи.

- `WaitAll(Task[])` - блокирует вызывающий поток до завершения всех задач в массиве.

- `WaitAny(Task[])` - блокирует вызывающий поток до завершения хотя бы одной из задач в массиве.

6. **Пример синхронного запуска Task:**

```csharp

Task task = new Task(() => Console.WriteLine("Task executed."));

task.RunSynchronously();

```

7. **Создание задачи с возвратом результата:**

```csharp

Task<int> taskWithResult = Task.Run(() => 42);

```

8. **Обработка исключения при выполнении Task:**

```csharp

try

{

Task.Run(() => { /\* код, который может вызвать исключение \*/ }).Wait();

}

catch (AggregateException ex)

{

// Обработка исключения

}

```

9. **CancellationToken и отмена выполнения задач:**

- `CancellationToken` позволяет передавать сигнал о том, что задача должна быть отменена. Можно использовать метод `ThrowIfCancellationRequested()` или проверять свойство `IsCancellationRequested`.

10. **Задача продолжения (continuation task):**

```csharp

Task<int> originalTask = Task.Run(() => 42);

Task continuationTask = originalTask.ContinueWith(prevTask => Console.WriteLine(prevTask.Result));

```

11. **Объект ожидания при создании задач продолжения:**

- Метод `ContinueWith` возвращает `Task`, который представляет продолжение предыдущей задачи. Можно использовать его для ожидания завершения продолжения.

12. **Назначение класса System.Threading.Tasks.Parallel:**

- `Parallel` предоставляет простой способ распараллеливания циклов и операций в .NET, автоматически управляя распределением работы по потокам.

13. **Пример задачи с Parallel.For(int, int, Action<int>):**

```csharp

Parallel.For(0, 10, i => Console.WriteLine(i));

```

14. **Пример задачи с Parallel.ForEach:**

```csharp

var data = Enumerable.Range(1, 10);

Parallel.ForEach(data, item => Console.WriteLine(item));

```

15. **Пример с Parallel.Invoke():**

```csharp

Parallel.Invoke(

() => Console.WriteLine("Task 1"),

() => Console.WriteLine("Task 2"),

() => Console.WriteLine("Task 3")

);

```

16. **Отмена параллельных операций с использованием CancellationToken:**

- Создайте `CancellationTokenSource`, передайте его в методы `Parallel` и используйте методы `Cancel` и `Token` для управления отменой.

17. **Использование BlockingCollection<T>:**

- `BlockingCollection<T>` является удобной оберткой вокруг `Queue<T>` или `Stack<T>`, обеспечивая потокобезопасный доступ и автоматическую блокировку при чтении/записи.

18. **Асинхронное выполнение метода с использованием async и await:**

```csharp

async Task<int> MyAsyncMethod()

{

// асинхронная операция

return await SomeAsyncOperation();

}

```