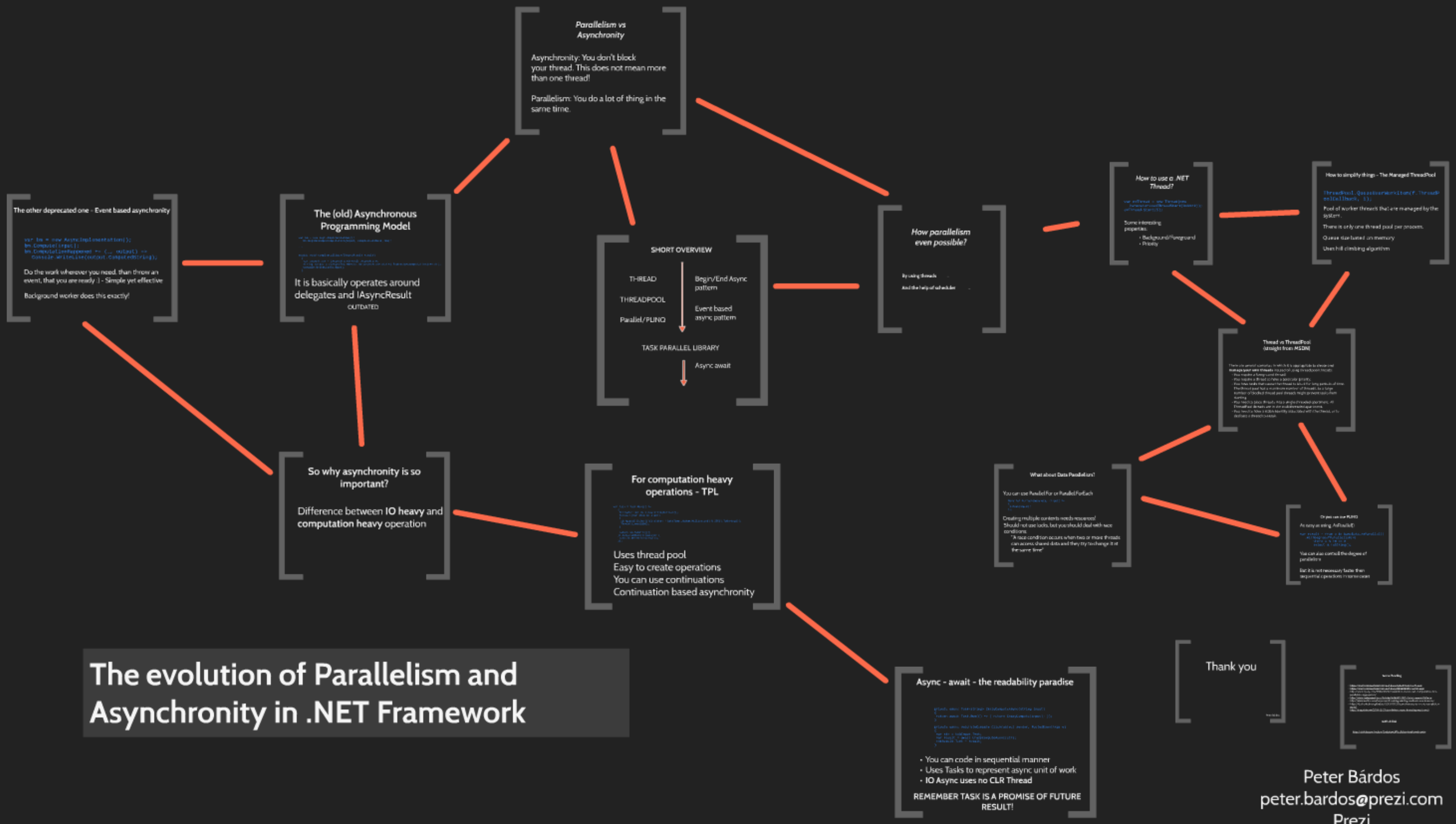
The evolution of Parallelism and Asynchronity in .NET Framework



Parallelism vs Asynchronity

Asynchronity: You don't block your thread. This does not mean more than one thread!

Parallelism: You do a lot of thing in the same time.

The other deprecated one - Event based asynchronity

```
var bm = new AsyncImplementation();
bm.Compute(input);
bm.ComputationHappened += (_, output) =>
  Console.WriteLine(output.ComputedString);
```

Do the work wherever you need, than throw an event, that you are ready :) - Simple yet effective

Background worker does this exactly!

The (old) Asynchronous Programming Model

It is basically operates around delegates and IAsyncResult

OUTDATED

SHORT OVERVIEW

THREAD

THREADPOOL

Parallel/PLINQ

Begin/End Async pattern

Event based async pattern

TASK PARALLEL LIBRARY

Async await

How parallelism even possible?

By using threads

And the help of scheduler

How to use a .NET Thread?

Some interesting properties:

- Background/Foreground
- Priority

How to simplify things - The Managed ThreadPool

```
ThreadPool.QueueUserWorkItem(f.ThreadPoolCallback, 1);
```

Pool of worker threads that are managed by the system.

There is only one thread pool per process.

Queue size based on memory

Uses hill climbing algorithm

Thread vs ThreadPool
(straight from MSDN)

There are several scenarios in which it is appropriate to create and **manage your own threads** instead of using thread pool threads:

- You require a foreground thread.
- You require a thread to have a particular priority.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

What about Data Parallelism?

You can use Parallel.For or Parallel.ForEach

Creating multiple contexts needs resources!
Should not use locks, but you should deal with race conditions.

"A race condition occurs when two or more threads can access shared data and they try to change it at the same time"

Or you can use PLINQ

As easy as using .AsParallel()

You can also controll the degree of parallelism

But it is not necessary faster then sequential operations in some cases

So why asynchronity is so important?

Difference between **IO heavy** and **computation heavy** operation

For computation heavy operations - TPL

Uses thread pool
Easy to create operations
You can use continuations
Continuation based asynchrony

Async - await - the readability paradise

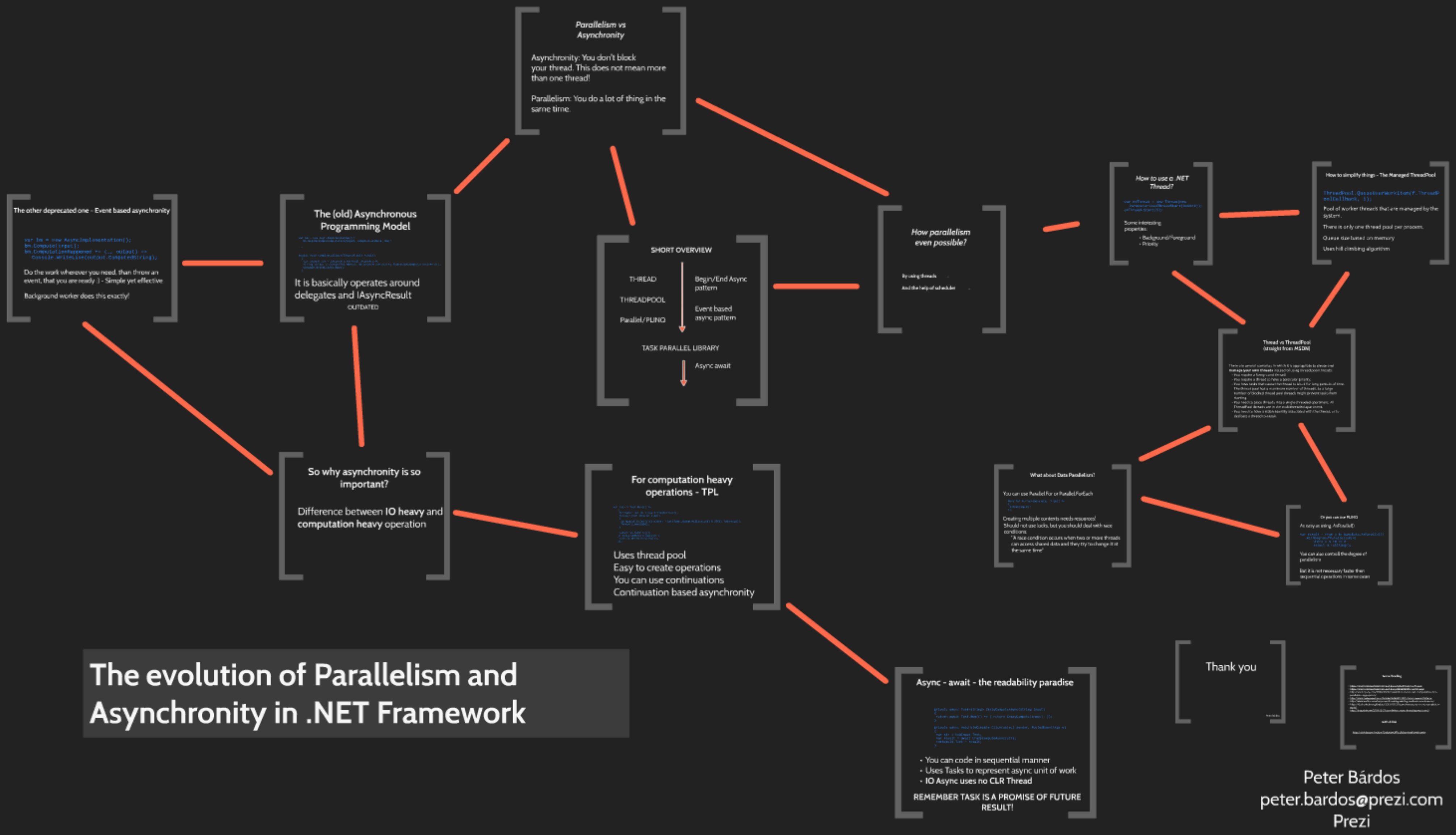
- You can code in sequential manner
- Uses Tasks to represent async unit of work
- IO Async uses no CLR Thread

REMEMBER TASK IS A PROMISE OF FUTURE RESULT!

Thank you

Further Reading

Peter Bárdos
peter.bardos@prezi.com
Prezi



Parallelism vs Asynchronity

Asynchronity: You don't block your thread. This does not mean more than one thread!

Parallelism: You do a lot of thing in the same time.

The other deprecated one - Event based asynchronity

```
var bm = new AsyncImplementation();
bm.Compute(input);
bm.ComputationHappened += (_, output) =>
  Console.WriteLine(output.ComputedString);
```

Do the work wherever you need, than throw an event, that you are ready :) - Simple yet effective

Background worker does this exactly!

The (old) Asynchronous Programming Model

It is basically operates around delegates and IAsyncResult

OUTDATED

SHORT OVERVIEW

THREAD

THREADPOOL

Parallel/PLINQ

Begin/End Async pattern

Event based async pattern

TASK PARALLEL LIBRARY

Async await

How parallelism even possible?

By using threads

And the help of scheduler

How to use a .NET Thread?

Some interesting properties:

- Background/Foreground
- Priority

How to simplify things - The Managed ThreadPool

```
ThreadPool.QueueUserWorkItem(f.ThreadPoolCallback, 1);
```

Pool of worker threads that are managed by the system.

There is only one thread pool per process.

Queue size based on memory

Uses hill climbing algorithm

Thread vs ThreadPool
(straight from MSDN)

There are several scenarios in which it is appropriate to create and **manage your own threads** instead of using thread pool threads:

- You require a foreground thread.
- You require a thread to have a particular priority.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

What about Data Parallelism?

You can use Parallel.For or Parallel.ForEach

Creating multiple contexts needs resources!
Should not use locks, but you should deal with race conditions:
"A race condition occurs when two or more threads can access shared data and they try to change it at the same time"

Or you can use PLINQ

As easy as using .AsParallel()

You can also controll the degree of parallelism

But it is not necessary faster then sequential operations in some cases

So why asynchronity is so important?

Difference between **IO heavy** and **computation heavy** operation

For computation heavy operations - TPL

Uses thread pool
Easy to create operations
You can use continuations
Continuation based asynchronity

Async - await - the readability paradise

- You can code in sequential manner
- Uses Tasks to represent async unit of work
- IO Async uses no CLR Thread

REMEMBER TASK IS A PROMISE OF FUTURE RESULT!

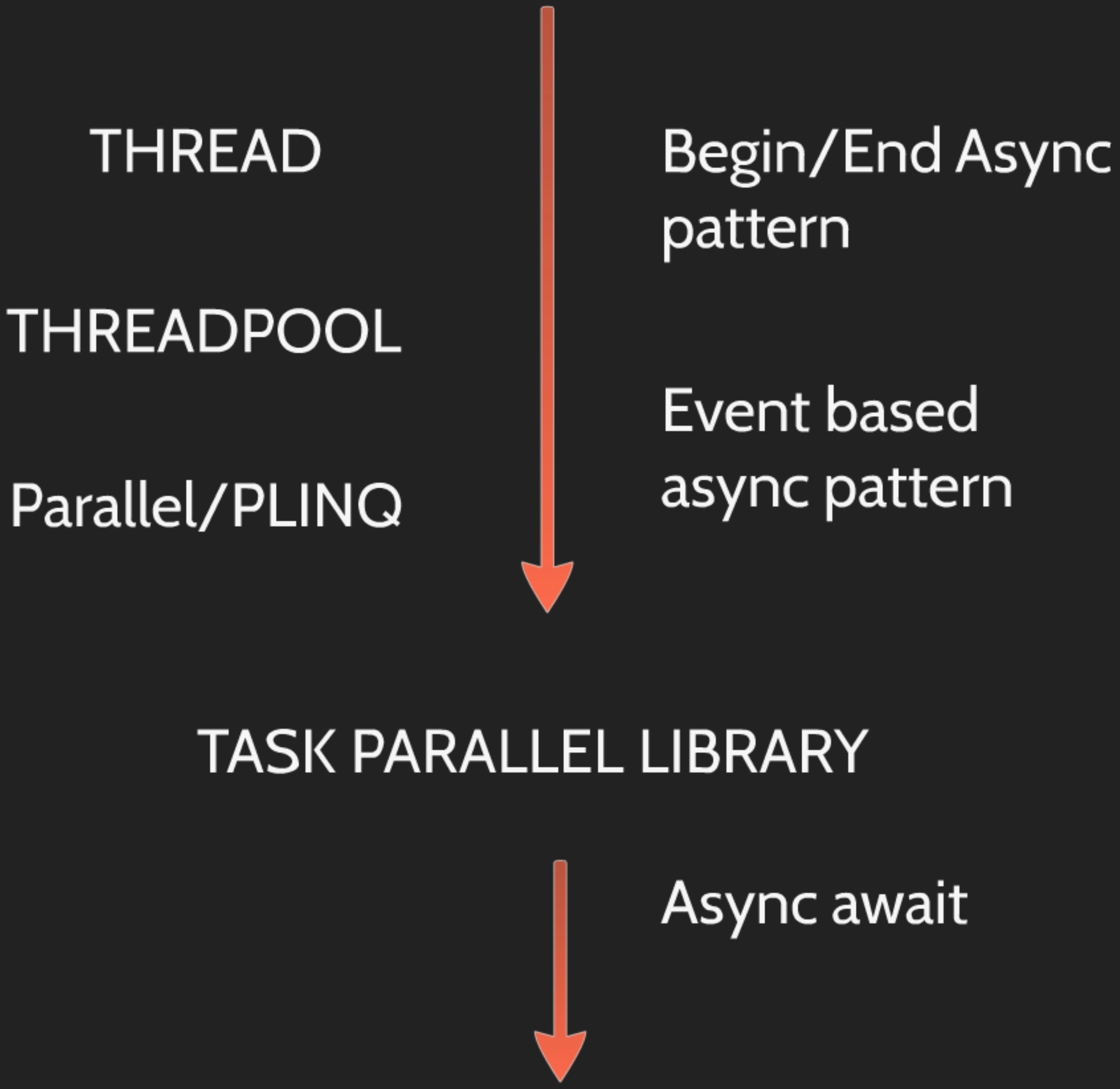
Thank you

[illegible]

The evolution of Parallelism and Asynchronity in .NET Framework

Peter Bárdos
peter.bardos@prezi.com
Prezi

SHORT OVERVIEW



THREAD

THREADPOOL

Parallel/PLINQ

Begin/End Async pattern

Event based async pattern

TASK PARALLEL LIBRARY

Async await

Parallelism vs Asynchronity

Asynchronity: You don't block your thread. This does not mean more than one thread!

Parallelism: You do a lot of thing in the same time.

How parallelism even possible?

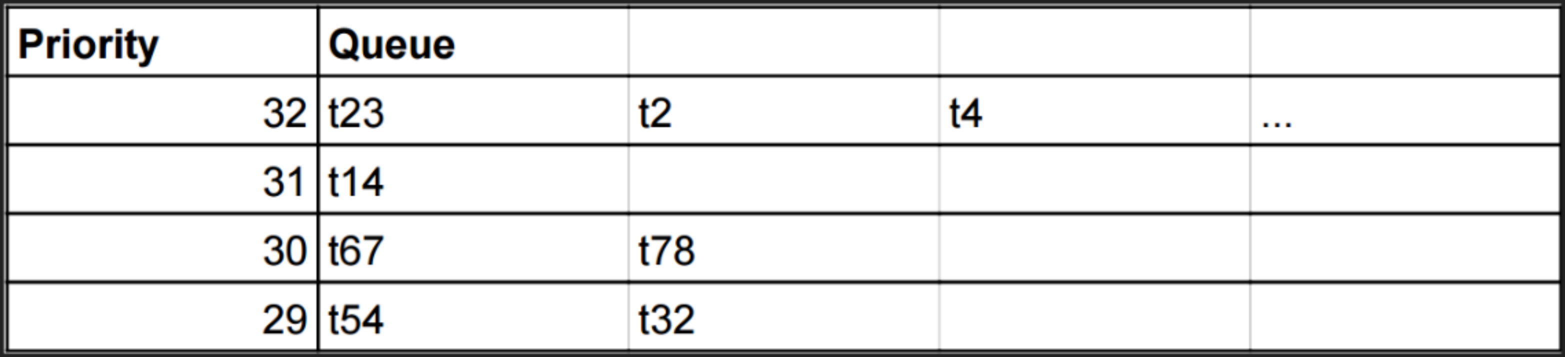
By using threads

And the help of scheduler

WHAT IS A THREAD?

A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler

PRIORITY PREEMPTIVE SCHEDULING



Priority	Queue			
32	t23	t2	t4	...
31	t14			
30	t67	t78		
29	t54	t32		

How to use a .NET Thread?

```
var exThread = new Thread(new
  ParameterizedThreadStart(DoWork));
exThread.Start(5);
```

Some interesting properties:

- Background/Foreground
- Priority

How to simplify things - The Managed ThreadPool

```
ThreadPool.QueueUserWorkItem(f.ThreadPoolCallback, i);
```

Pool of worker threads that are managed by the system.

There is only one thread pool per process.

Queue size based on memory

Uses hill climbing algorithm

Thread vs ThreadPool
(straight from MSDN)

There are several scenarios in which it is appropriate to create and **manage your own threads** instead of using thread pool threads:

- You require a foreground thread.
- You require a thread to have a particular priority.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

What about Data Parallelism?

You can use Parallel.For or Parallel.ForEach

```
Parallel.ForEach(baseData, (input) =>
{
 l.Push(input);
});
```

Creating multiple contexts needs resources!

Should not use locks, but you should deal with race conditions:

> "A race condition occurs when two or more threads can access shared data and they try to change it at the same time"

Or you can use PLINQ

As easy as using .AsParallel()

```
var result = from u in baseData.AsParallel()
   .WithDegreeOfParallelism(4)
       where u % 10 == 0
       select u.ToString();
```

You can also controll the degree of parallelism

But it is not necessary faster then sequential operations in some cases

Parallelism vs Asynchronity

Asynchronity: You don't block your thread. This does not mean more than one thread!

Parallelism: You do a lot of thing in the same time.

The (old) Asynchronous Programming Model

```
var bm = new AsyncImplementation();
   bm.BeginComlpexComputation(input, ComplexCallback, bm);

...


static void ComplexCallback(IAsyncResult result)
  {
   var asyncState = (AsyncState)result.AsyncState;
   string output = ((AsyncImplementation)asyncState.State).EndComlpexComputation(result);
   Console.WriteLine(output);
  }
```

It is basically operates around delegates and IAsyncResult

OUTDATED

The other deprecated one - Event based asynchronity

```
var bm = new AsyncImplementation();
bm.Compute(input);
bm.ComputationHappened += (_, output) =>
  Console.WriteLine(output.ComputedString);
```

Do the work wherever you need, than throw an event, that you are ready :) - Simple yet effective

Background worker does this exactly!

So why asynchronity is so important?

Difference between **IO heavy** and **computation heavy** operation

For computation heavy operations - TPL

```
var task = Task.Run(() =>
   {
    StringBuilder sb = new StringBuilder();
    foreach (var item in input)
    {
     sb.Append(((char)(((int)item * DateTime.UtcNow.Millisecond) % 255)).ToString());
     Thread.Sleep(1000);
    }

    return sb.ToString();
   }).ContinueWith((result)=> {
    Console.WriteLine(result);
   });
```

Uses thread pool

Easy to create operations

You can use continuations

Continuation based asynchronity

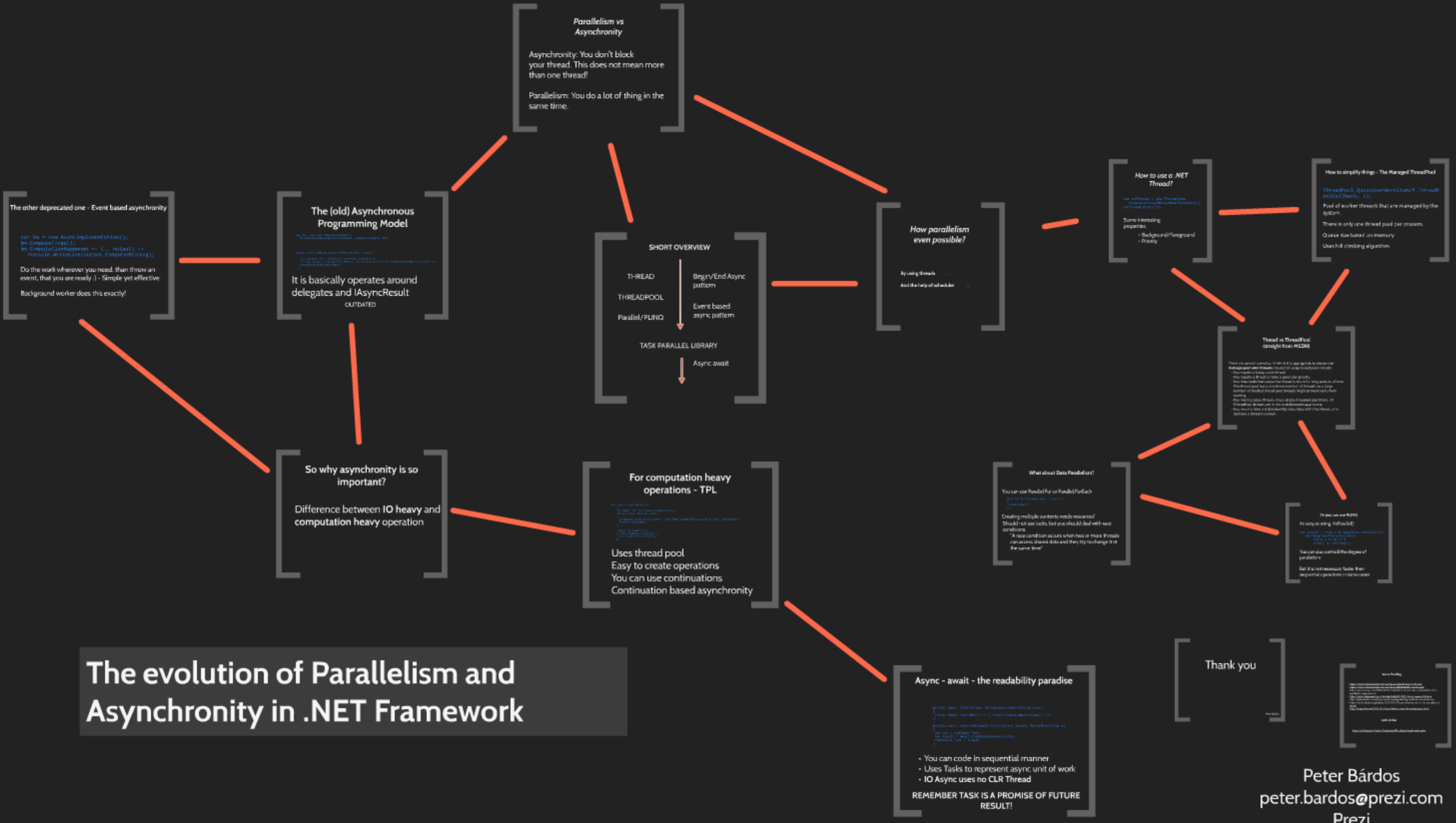
Async - await - the readability paradise

```
private async Task<string> CrazyComputeAsync(string input)
{
 return await Task.Run(() => { return CrazyCompute(input); });
}

private async void btnCompute_Click(object sender, RoutedEventArgs e)
{
 var str = txbInput.Text;
 var result = await CrazyComputeAsync(str);
 txtResult.Text = result;
}
```

- You can code in sequential manner
- Uses Tasks to represent async unit of work
- **IO Async uses no CLR Thread**

REMEMBER TASK IS A PROMISE OF FUTURE RESULT!



The evolution of Parallelism and Asynchronity in .NET Framework

Parallelism vs Asynchronity

Asynchronity: You don't block your thread. This does not mean more than one thread!

Parallelism: You do a lot of thing in the same time.

The other deprecated one - Event based asynchronity

```
var bm = new AsyncImplementation();
bm.Compute(input);
bm.ComputationHappened += (_, output) =>
   Console.WriteLine(output.ComputedString);
```

Do the work wherever you need, than throw an event, that you are ready :) - Simple yet effective

Background worker does this exactly!

The (old) Asynchronous Programming Model

```
[illegible]
```

It is basically operates around delegates and IAsyncResult

OUTDATED

SHORT OVERVIEW

THREAD

THREADPOOL

Parallel/PLINQ

Begin/End Async pattern

Event based async pattern

TASK PARALLEL LIBRARY

Async await

How parallelism even possible?

By using threads

And the help of scheduler

How to use a .NET Thread?

```
[illegible]
```

Some interesting properties:

- Background/Foreground
- Priority

How to simplify things - The Managed ThreadPool

```
ThreadPool.QueueUserWorkItem(f.ThreadPoolCallback, i);
```

Pool of worker threads that are managed by the system.

There is only one thread pool per process.

Queue size based on memory

Uses hill climbing algorithm

Thread vs ThreadPool (straight from MSDN)

There are several scenarios in which it is appropriate to create and **manage your own threads** instead of using thread pool threads:

- You require a foreground thread.
- You require a thread to have a particular priority.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

What about Data Parallelism?

You can use Parallel.For or Parallel.ForEach

```
[illegible]
```

Creating multiple contexts needs resources!

Should not use locks, but you should deal with race conditions.

"A race condition occurs when two or more threads can access shared data and they try to change it at the same time"

Or you can use PLINQ

As easy as using .AsParallel()

```
[illegible]
```

You can also controll the degree of parallelism

But it is not necessary faster then sequential operations in some cases

So why asynchronity is so important?

Difference between **IO heavy** and **computation heavy** operation

For computation heavy operations - TPL

```
[illegible]
```

Uses thread pool
Easy to create operations
You can use continuations
Continuation based asynchronity

Async - await - the readability paradise

```
[illegible]
```

- You can code in sequential manner
- Uses Tasks to represent async unit of work
- IO Async uses no CLR Thread

REMEMBER TASK IS A PROMISE OF FUTURE RESULT!

Thank you

Some Reading

[illegible]

Peter Bárdos
peter.bardos@prezi.com
Prezi

Thank you

Peter Bárdos

Some Reading

- https://msdn.microsoft.com/en-us/library/aa645740(v=vs.71).aspx
- https://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx
- http://reedcopsey.com/2010/01/22/parallelism-in-net-part-4-imperative-data-parallelism-aggregation/
- http://www.codeproject.com/Articles/646239/NET-Asynchronous-Patterns
- http://www.matlus.com/iasyncresult-making-existing-methods-asnchronous/
- https://dschenkelman.github.io/2013/10/29/asynchronous-io-in-c-io-completion-ports/
- http://blog.slaks.net/2014-12-23/parallelism-async-threading-explained/

GitHub link

https://github.com/melorn/EvolutionOfParallelismAndAsynchronity