braspag
uma empresa Cielo.

Programação Assíncrona e Paralelismo no WebAPI

Johnathan Fercher

02/02/2018

Sumário

1. Introdução

2. Programação Síncrona e Assíncrona em C#

3. Requisições no WebAPI

4. Paralelismo x Concorrência

5. Paralelismo de Dados no C#

6. Paralelismo de Tarefas em C#

7. Um Último Hack

8. Conclusões

Introdução

Objetivo

- Apresentar as principais formas de aprimorar a performance de execução de requisições no WebAPI;

Definições

- **Programação Assíncrona** tem o propósito de liberar threads em operações bloqueantes;
- **Programação Paralela** tem o propósito de possibilitar a execução de tarefas ao mesmo tempo;

Problema assíncrono

- Consumir uma API;
- Operações de CRUD em banco de dados;
- Quaisquer outras operações de I/O;

Problema paralelo

- **Em um jogo:** Uma thread é responsável por obter os comandos do Joystick e outras N são responsáveis por controlar a renderização de objetos, comandos de adversários e etc;
- **Em um algoritmo de reconhecimento facial:** Pode-se dividir uma imagem em quadrantes, onde N threads serão responsáveis por aplicar filtros nas secções;

Programação Síncrona e Assíncrona em C#

Sintaxe de um código síncrono

```
41          public void Execute(){
42              var value = Method();
43          }
```

- 1 - Ao chamar **Method()**, a execução é movida para dentro da função.
- 2 - Dentro de **Method()**, quando ocorre uma chamada bloqueante, a linguagem realiza um espera ocupada até que a operação de I/O acabe;
- 3 - Quando a operação de I/O acaba, a execução retorna para **Execute()**;

Sintaxe de um código assíncrono

```
41          public async Task<T> Execute(){
42              T value = await MethodAsync();
43              return value;
44          }
```

- 1 - Ao chamar **MethodAsync()**, a execução é movida para dentro da função.
- 2 - Dentro de **MethodAsync()**, quando ocorre uma chamada bloqueante, a execução retorna para **Execute()**;
- 3 - Paralelamente, **Execute()** pode realizar trabalho enquanto a chamada assíncrona espera a realização de operações de I/O;
- 4 - Quando a operação assíncrona acaba, um evento é emitido e a linguagem sabe obter o resultado do método;

Forçando execução assíncrona de um método síncrono

```
41          public async Task<T> MethodAsync(){
42            return await Task.Run(() => {
43                return Method();
44              });
45          }
```

Nota Importante: Não existe benefício de escalabilidade para essa transformação.

Transformando código assíncrono em síncrono

```
41          public T Method(){
42            return MethodAsync().Wait();
43          }
```

Nota Importante: Somente utilize Wait() se o escopo que utiliza o código assíncrono PRECISAR ser síncrono.

Ex: Startup de aplicações;

Transformando código assíncrono em síncrono

```
23              public T Execute(){
24                return MethodAsync().Result;
25              }
```

Nota Importante: Jamais utilize Result no WebAPI. Perigo de deadlock.

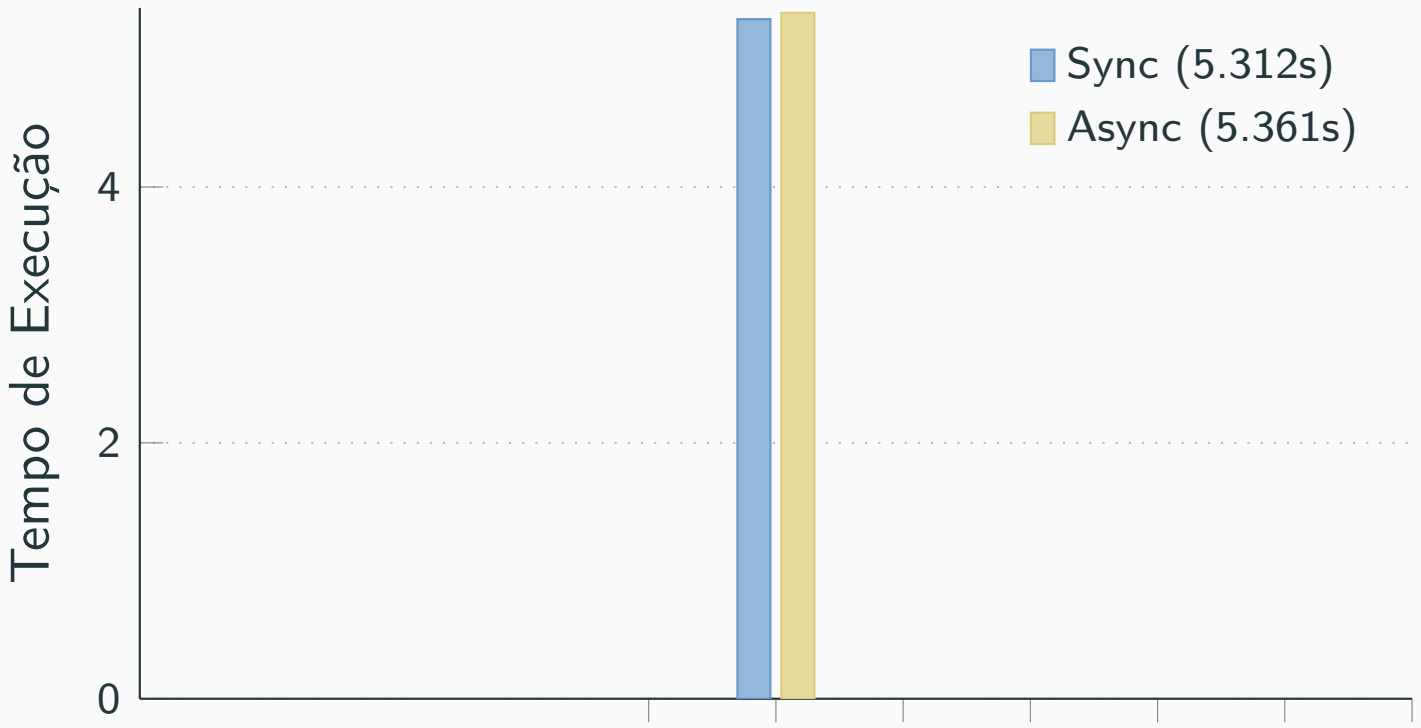
Transformando código assíncrono em síncrono

```
23              public async Task<T> Execute(){
24                return await MethodAsync().ConfigureAwait(false);
25              }
```

Exemplos para Benchmark

Sync x Async

Benchmark



- Sync: Síncrono;
- Async: Assíncrono;

Fui tapeado?

Por que não houve ganhos de performance?

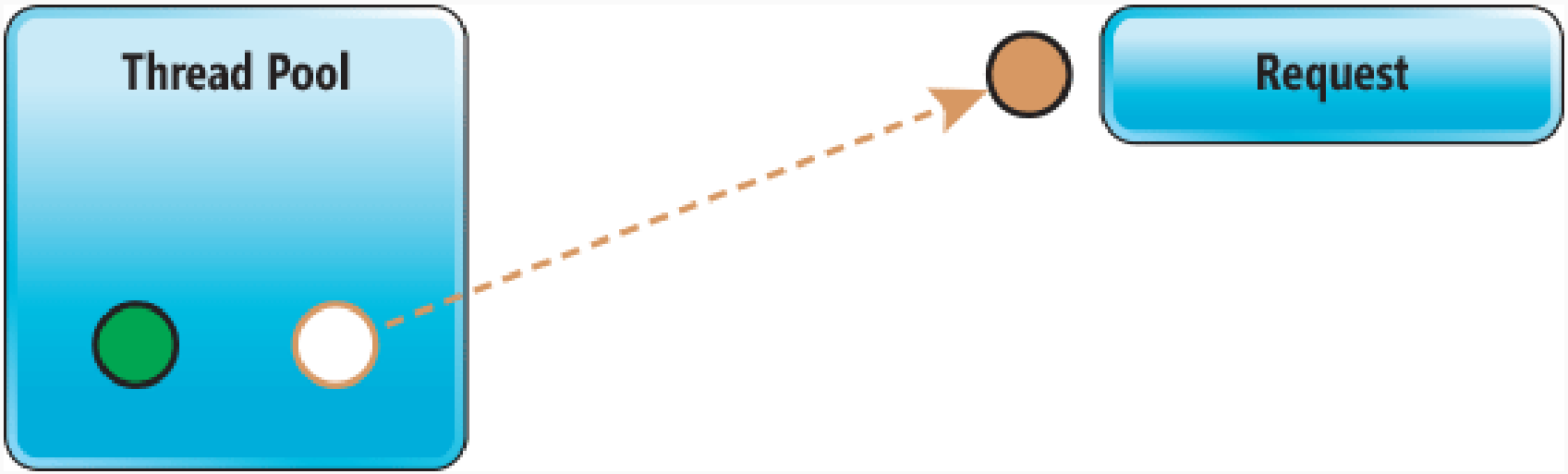
Programação Assíncrona apenas é responsável por não bloquear a execução.

Então para que vou usar isso?

Depende da aplicação.

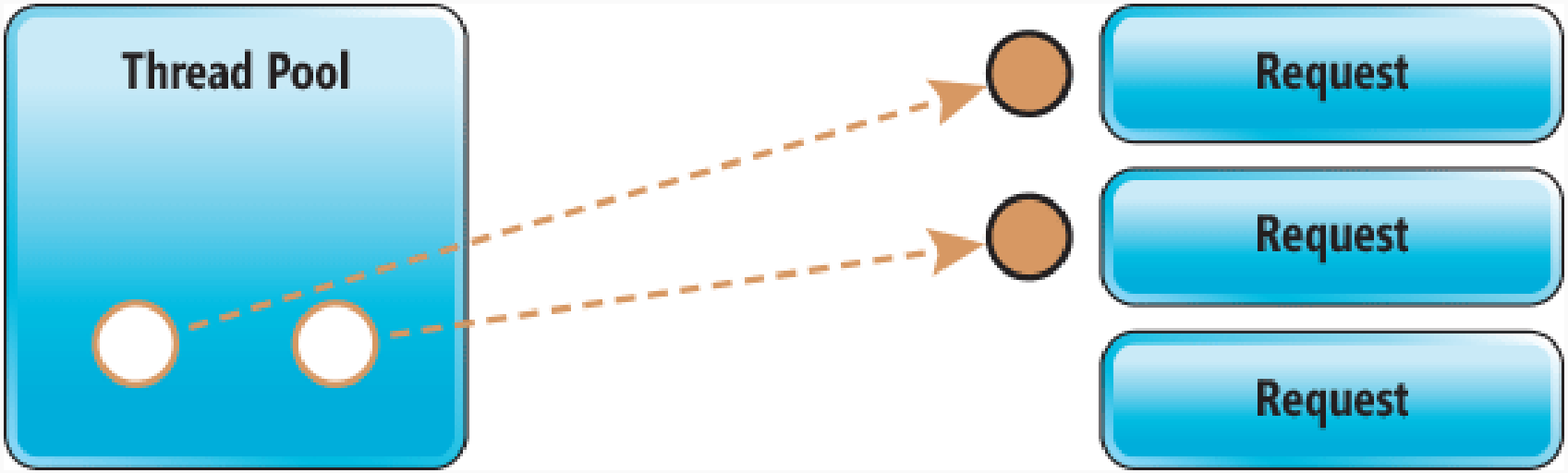
Requisições no WebAPI

Uma requisição síncrona no WebAPI



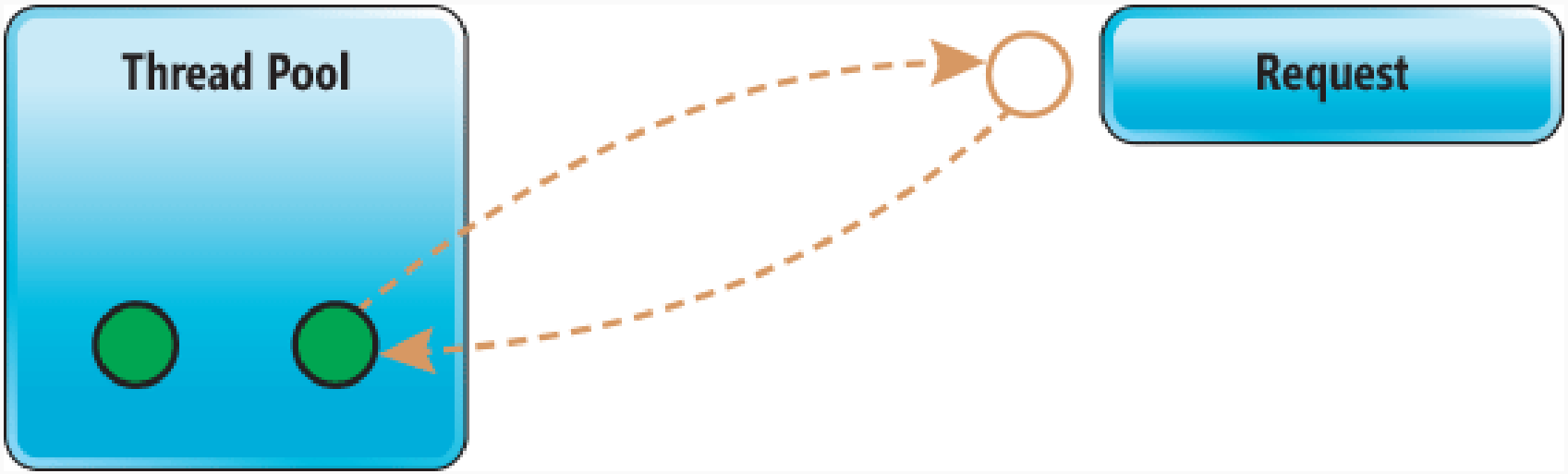
- Uma thread livre e outra thread bloqueada realizando **nada**;

Três requisições síncronas no WebAPI



- Duas threads bloqueadas realizando **nada** e uma requisição em espera;

Uma requisição assíncrona no WebAPI



- Thread realiza uma requisição bloqueante e retorna para a espera;

Ganhos

- Redução de custos em relação à memória;
- Redução do tempo de CPU desperdiçado;
- **Ganhos de performance** dependendo do problema;

Task.Run

```
41          public async Task<T> MethodAsync(){
42            return await Task.Run(() => {
43                return Method();
44              });
45          }
```

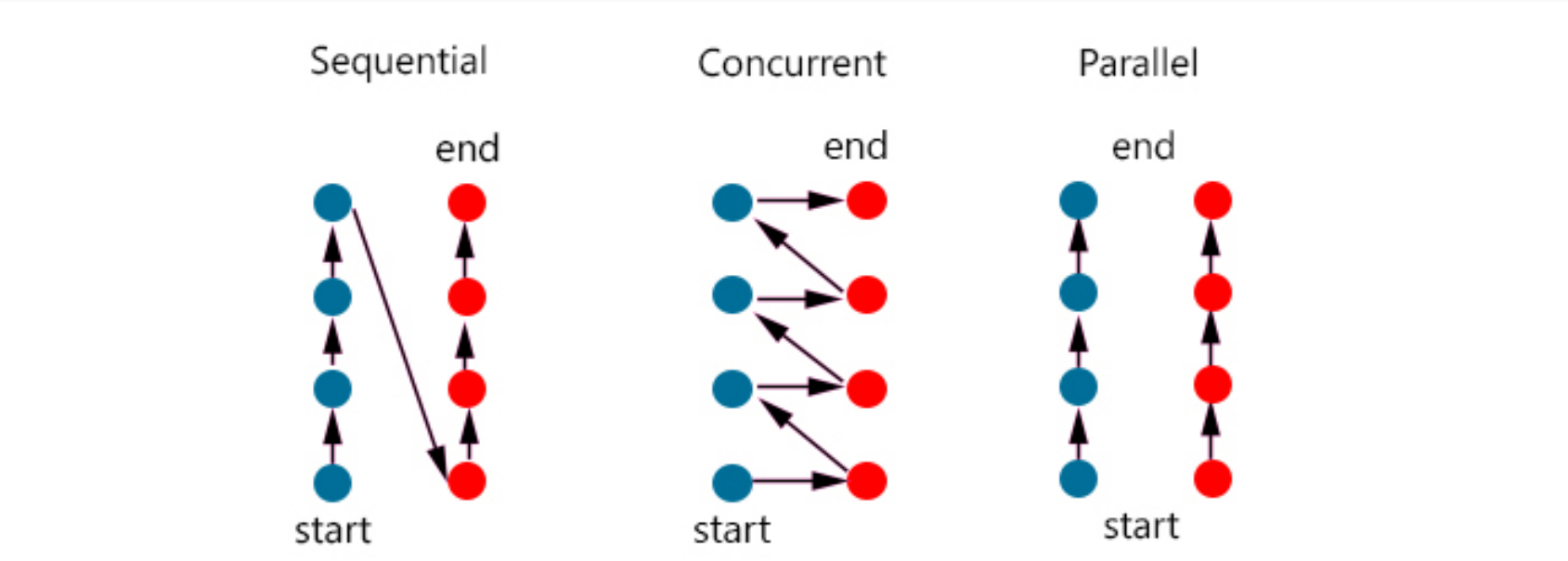
Por que Task.Run() não ajuda na escalabilidade?

- 1 - Ao chamar **await Task.Run()**, a execução é movida para dentro da função.
- 2 - Dentro de **MethodAsync()**, quando ocorre uma chamada bloqueante, a execução retorna;
- 3 - Paralelamente, **Execute()** pode realizar trabalho enquanto a chamada assíncrona espera a realização de operações de I/O;
- 4 - Em outra Task, MethodAsync() realiza chamada esperada;

Paralelismo x Concorrência

Paralelismo X Concorrência

- **Paralelismo:** Duas tarefas são executadas simultaneamente em diferentes processadores ou núcleos;
- **Concorrência:** Duas tarefas estão em progresso ao mesmo tempo, de forma alternada em um processador ou núcleo;



Paralelismo X Concorrência

- **Em um processador de 4 núcleos:**
  - Executando 4 threads. Pode-se alocar 1 thread por núcleo;
  - Executadas 5 threads. Pelo menos 2 threads vão ser alternadas em um único núcleo;
- **Vantagens:**
  - Execução Simultânea;
  - Execução Simultânea e aumento da frequência com que as tarefas são alocadas na CPU em um SO de tempo compartilhado;

Tipos de Paralelismo

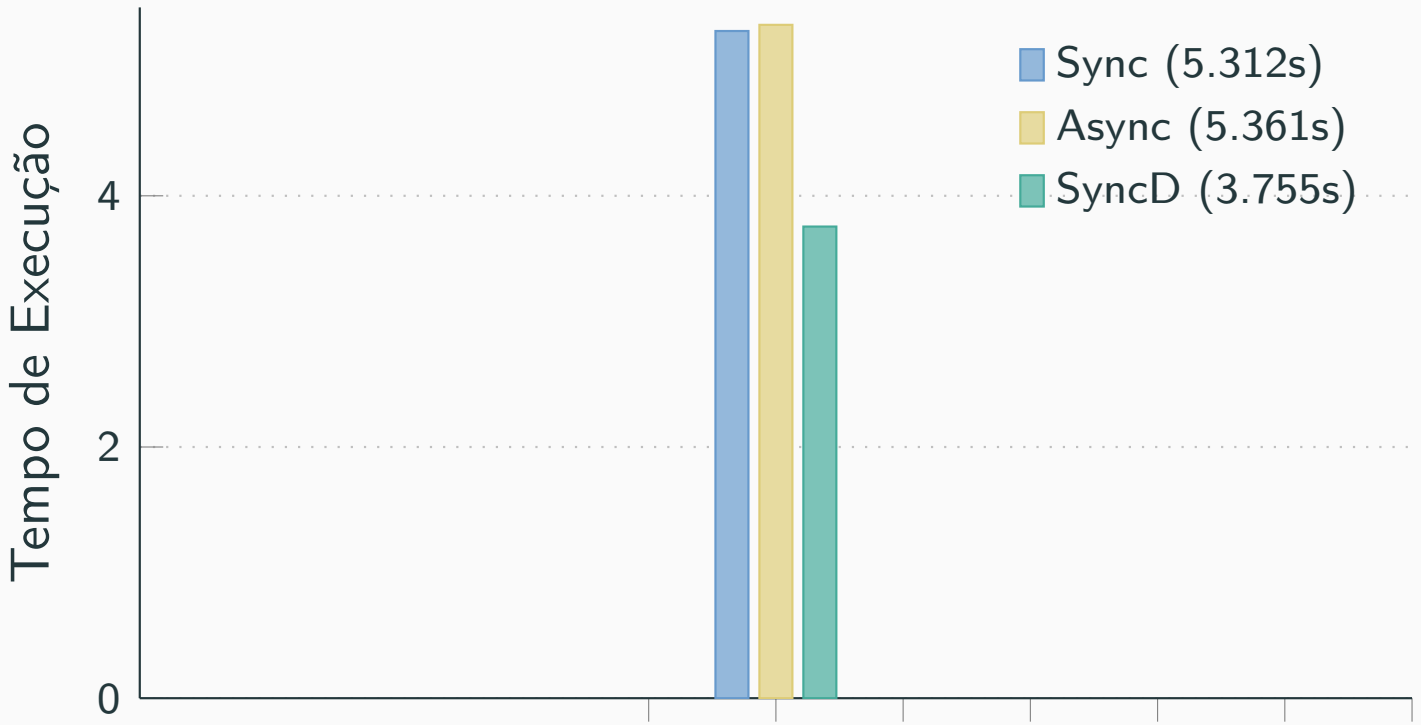
- **Paralelismo de Dados:** A mesma instrução é executada em diferentes threads.
  - Ex: Enquanto uma thread itera sobre a parte inicial de uma lista, outra itera sobre a parte final;
- **Paralelismo de Tarefas:** Diferentes instruções são executadas em várias threads;
  - Ex: Enquanto uma thread valida um token, outra busca credenciais no banco;

Paralelismo de Dados no C#

Exemplo para Benchmark

SyncDataParallel

Benchmark



- Sync: Síncrono;
- Async: Assíncrono;
- SyncD: Síncrono com Paralelismo de Dados;

Perguntas Frequentes

Posso aplicar isso em qualquer problema?

Depende da quantidade de transformação de dados envolvida.

Perguntas Frequentes

Quanto maior o número de threads melhor?

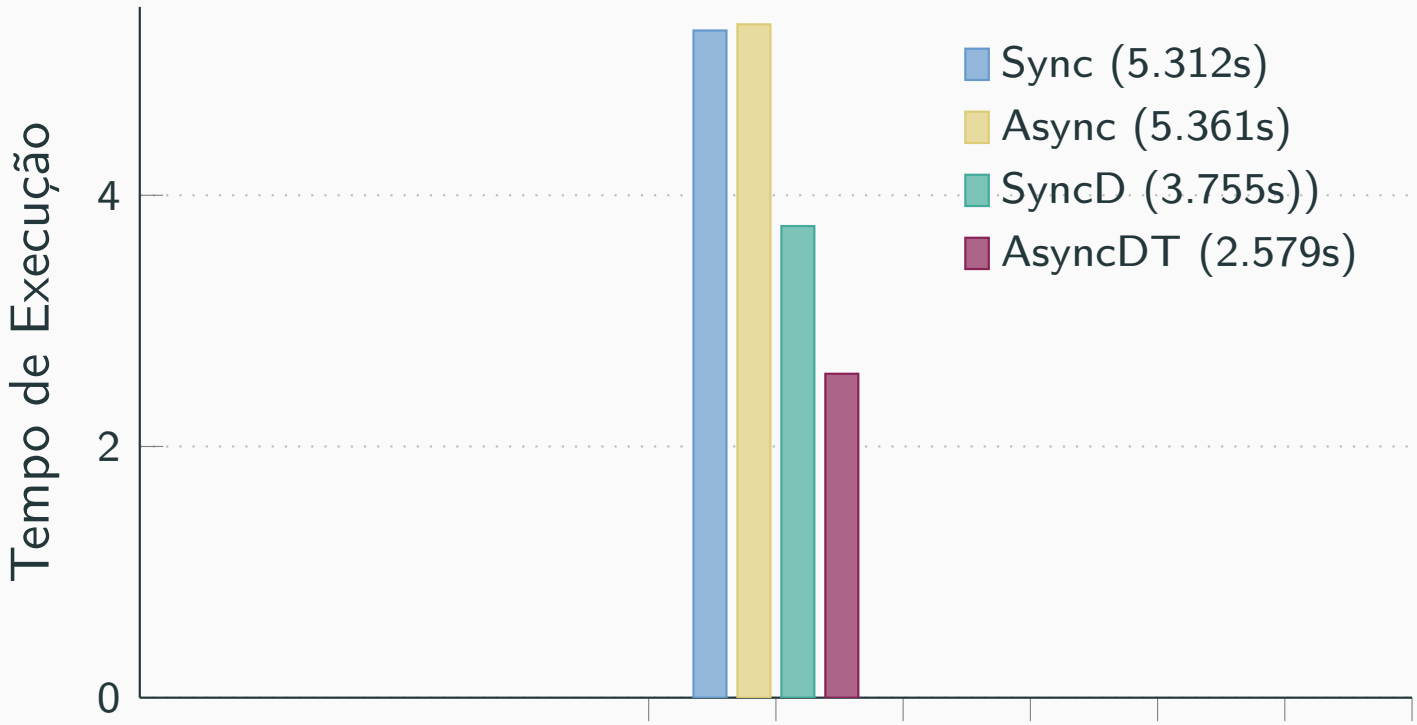
Existe um limiar onde a aplicação gasta mais tempo na comunicação entre threads do que o ganho de performance na execução paralela.

Paralelismo de Tarefas em C#

Exemplo para Benchmark

AsyncTaskAndDataParallel

Benchmark



- Sync: Síncrono;
- Async: Assíncrono;
- SyncD: Síncrono com Paralelismo de Dados;
- AsyncDT: Assíncrono com Paralelismo de Dados e de Tarefas;

E se...

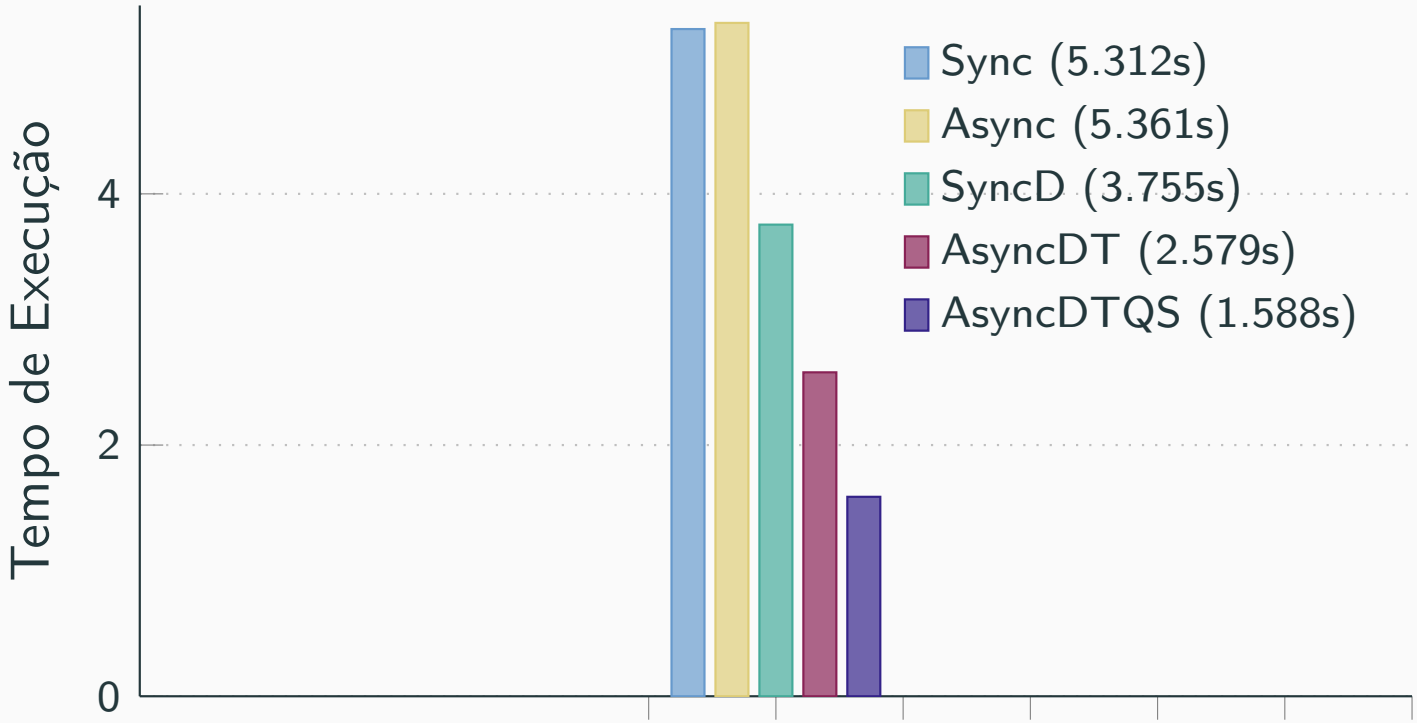
Paralelismo de dados dentro de paralelismo de tarefas...

Um Último Hack

Exemplo para Benchmark

AsyncTaskAndDataParallelQueueSave

Benchmark



- Sync: Síncrono;
- Async: Assíncrono;
- SyncD: Síncrono com Paralelismo de Dados;
- AsyncDT: Assíncrono com Paralelismo de Dados e Tarefas;
- AsyncDTQS: Assíncrono com Paralelismo de Dados e Tarefas com Fila de Execução;

Nota Importante

QueueWork != Async sem Await:

A second operation started on this context before a previous asynchronous operation completed. Use 'await' to ensure that any asynchronous operations have completed before calling another method on this context. Any instance members are not guaranteed to be thread safe.

Conclusões

Conclusões

- É possível criar métodos assíncronos e paralelos no WebAPI de forma simples e conveniente;
- Métodos assíncronos devem ser utilizados em APIs com finalidade de otimizar a utilização de recursos;
- É possível realizar paralelismo com métodos assíncronos;
- Paralelismo de dados deve ser utilizado em problemas onde a quantidade de transformações de dados é grande;

Obrigado