

Programação Assíncrona e Paralelismo no WebAPI

Johnathan Fercher

January 31, 2018

1. Introdução
2. Programação Síncrona e Assíncrona em C#
3. Paralelismo em C#
4. Programação Assíncrona e Paralela em C#
5. Conclusões

Introdução

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

- **Programação Assíncrona** tem o propósito de possibilitar a execução de tarefas demoradas sem o bloqueio da execução;
- **Programação Paralela** tem o propósito de possibilitar a execução de tarefas ao mesmo tempo;

Problema assíncrono

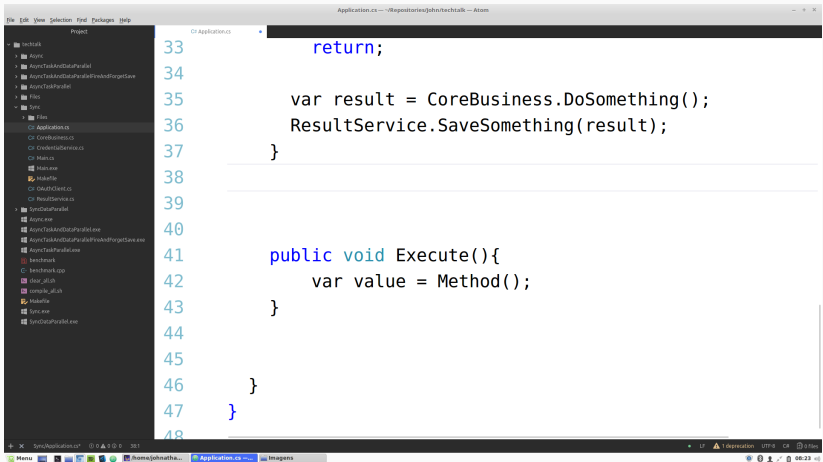
- Consumir uma API;
- Operações de CRUD em banco de dados;
- Quaisquer outras operações que demorem: **Treinamento de Inteligências Artificiais, Processamento de Imagens e Etc;**

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 robô:** Uma thread é responsável pela leitura de sensores e outras N são responsáveis por controlar motores, realizar comunicação com outros robôs 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;

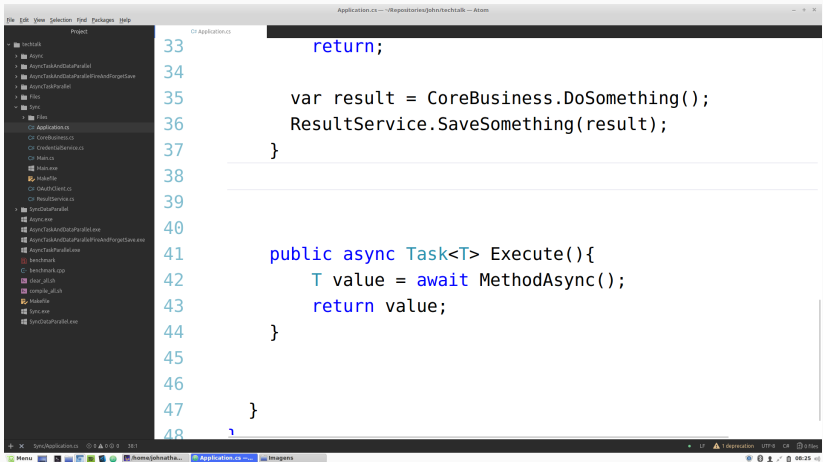
Programação Síncrona e Assíncrona em C#

Sintaxe de um código síncrono



```
33         return;  
34  
35         var result = CoreBusiness.DoSomething();  
36         ResultService.SaveSomething(result);  
37     }  
38  
39  
40  
41     public void Execute(){  
42         var value = Method();  
43     }  
44  
45  
46 }  
47  
48
```

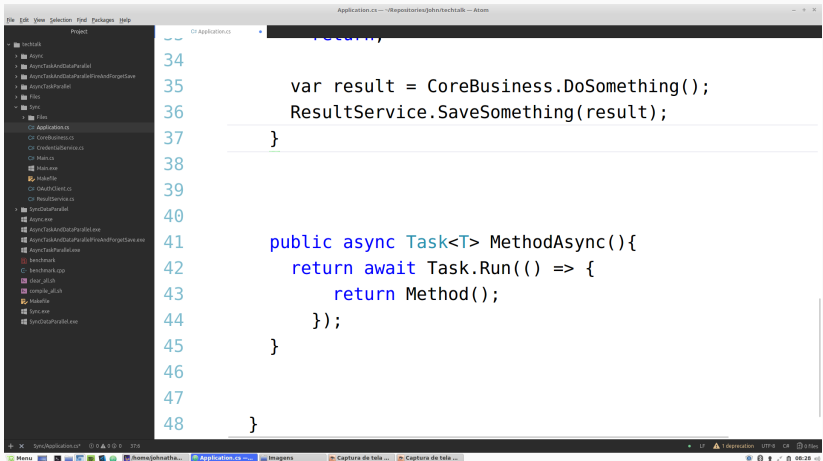
Sintaxe de um código assíncrono



The screenshot shows the Visual Studio IDE with a C# file named `Application.cs` open. The file explorer on the left shows a project structure with various files and folders. The code in the editor is as follows:

```
33         return;  
34  
35         var result = CoreBusiness.DoSomething();  
36         ResultService.SaveSomething(result);  
37     }  
38  
39  
40  
41     public async Task<T> Execute(){  
42         T value = await MethodAsync();  
43         return value;  
44     }  
45  
46  
47 }  
48
```

Transformando código síncrono em assíncrono

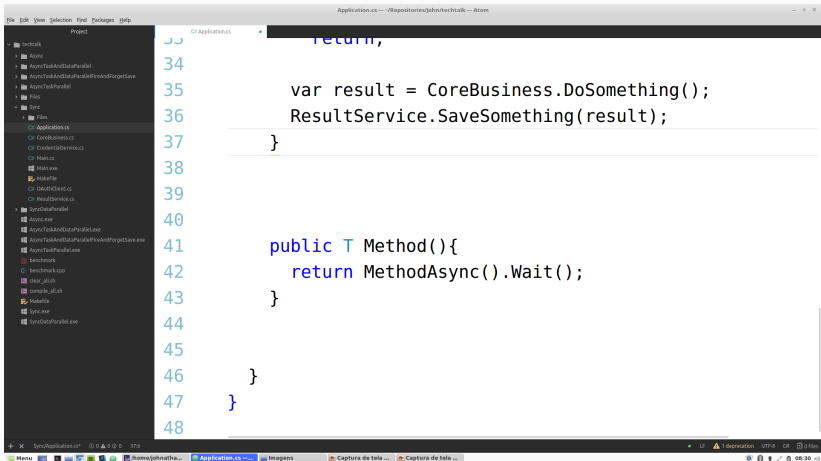


The screenshot shows a Visual Studio editor window titled 'Application.cs' with the following code:

```
34  
35     var result = CoreBusiness.DoSomething();  
36     ResultService.SaveSomething(result);  
37 }  
38  
39  
40  
41 public async Task<T> MethodAsync(){  
42     return await Task.Run(() => {  
43         return Method();  
44     });  
45 }  
46  
47  
48 }
```

The code is displayed in a dark-themed editor. The left sidebar shows a file explorer with a project structure including folders like 'bin', 'obj', and 'Files', and files like 'Application.cs', 'CoreBusiness.cs', 'CredentialService.cs', 'Main.cs', 'Main.exe', 'MainFile', 'OAuthClient.cs', 'ResultService.cs', 'SyncDataParallel', 'Async.exe', 'AsyncTaskAndDataParallel.exe', 'AsyncTaskAndDataParallelProAndForgetSave.exe', 'AsyncTaskParallel.exe', 'benchmark', 'benchmark.app', 'diag_alish', 'compil_alish', 'MainFile', 'Sync.exe', and 'SyncDataParallel.exe'. The status bar at the bottom shows 'SyncApplication.cs' with a line number of 375, and a taskbar at the very bottom with icons for 'Menu', 'Application.cs', 'Imagens', 'Captura de tela...', and 'Captura de tela...'.

Transformando código assíncrono em síncrono



The screenshot shows a Visual Studio code editor window titled "Application.cs - V:\Repositories\john\techtalk - Atom". The editor displays a C# file named "Application.cs" with the following code:

```
33 return,  
34  
35 var result = CoreBusiness.DoSomething();  
36 ResultService.SaveSomething(result);  
37 }  
38  
39  
40  
41 public T Method(){  
42     return MethodAsync().Wait();  
43 }  
44  
45  
46 }  
47 }  
48
```

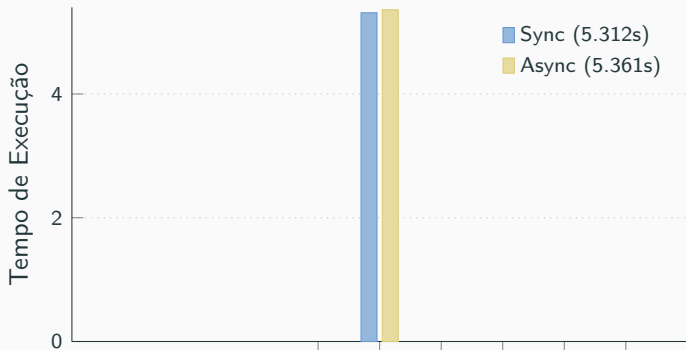
The code is structured as follows:

- Line 33: `return,`
- Line 34: (blank line)
- Line 35: `var result = CoreBusiness.DoSomething();`
- Line 36: `ResultService.SaveSomething(result);`
- Line 37: `}`
- Line 38: (blank line)
- Line 39: (blank line)
- Line 40: (blank line)
- Line 41: `public T Method(){`
- Line 42: `return MethodAsync().Wait();`
- Line 43: `}`
- Line 44: (blank line)
- Line 45: (blank line)
- Line 46: `}`
- Line 47: `}`
- Line 48: (blank line)

The left sidebar shows a project explorer with a tree view of files and folders. The bottom status bar indicates the file is "Application.cs" and the current line is 37.

Sync x Async

Benchmark



Fui tapeado?

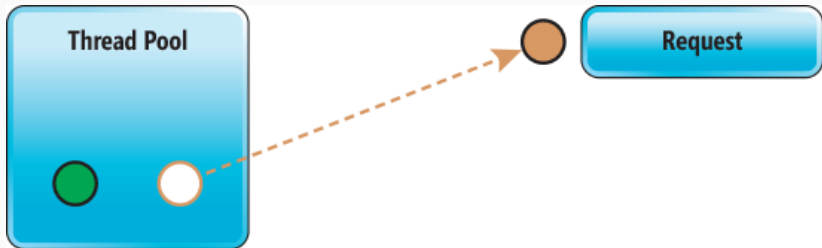
Por que não houve ganhos de performance?

Programação Assíncrona apenas é responsável por não bloquear a execução.

Então para que vou usar isso?

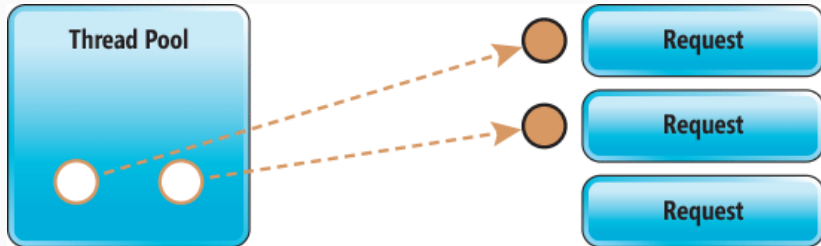
Depende da aplicação.

Uma requisição síncrona no WebAPI



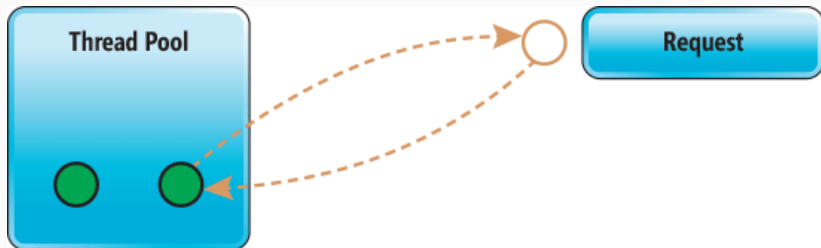
- Uma thread livre e outra thread bloqueada realizando **nada**;

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;

- Redução de custos em relação à memória;
- Redução do tempo de CPU desperdiçado;

Paralelismo em C#

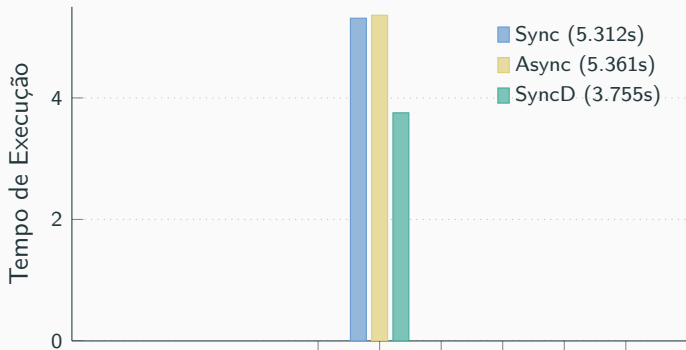
- **Paralelismo:** Duas tarefas são executadas simultaneamente.
 - Executadas em diferentes processadores;
 - Executadas em diferentes núcleos de um mesmo processador;
- **Concorrência:** Duas tarefas estão em progresso ao mesmo tempo;
 - Executadas de forma alternada em um mesmo núcleo;

- **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;

- **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;

SyncDataParallel

Benchmark



Problemas com Async

Posso aplicar isso em qualquer problema?

Depende da quantidade de transformação de dados envolvida.

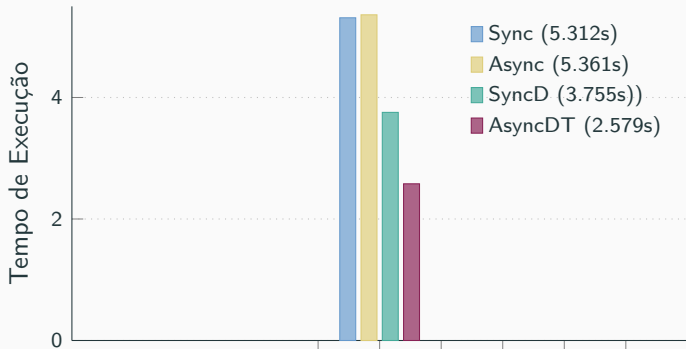
Quanto maior o número de threads melhor?

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

Programação Assíncrona e Paralela em C#

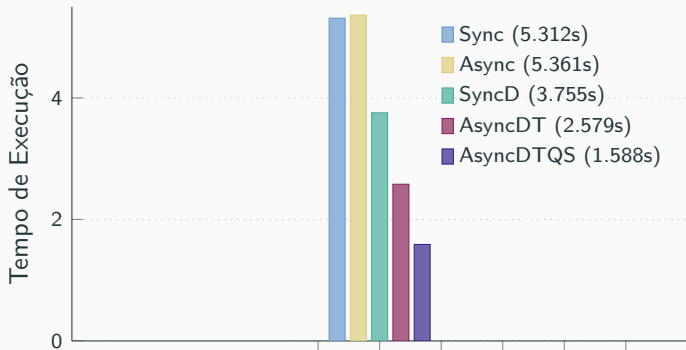
AsyncTaskAndDataParallel

Benchmark



AsyncTaskAndDataParallelQueueSave

Benchmark



QueueWork != Async sem Await

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;
- Paralelismo nunca fornece um ganho linear de performance e deve ser utilizado em problemas onde a quantidade de transformações de dados é grande;

Perguntas?