

Parallel execution in Microsoft .NET

Paolo Burgio

paolo.burgio@unimore.it





Outline

- › A few definitions
 - Brief introduction to .NET

- › Data parallelism in the .NET Framework
 - Understanding lambdas & closures

- › Task parallelism in the .NET Framework
 - “Asynchronous stack”

- › A real example
 - Web server



Microsoft .NET and the C# language

.NET framework

- › A complete, cross-platform programming framework completely integrated with Windows ecosystem
- › Integrated in Visual Studio (& VSCode)
- › .NET core as minimal/lightweight version



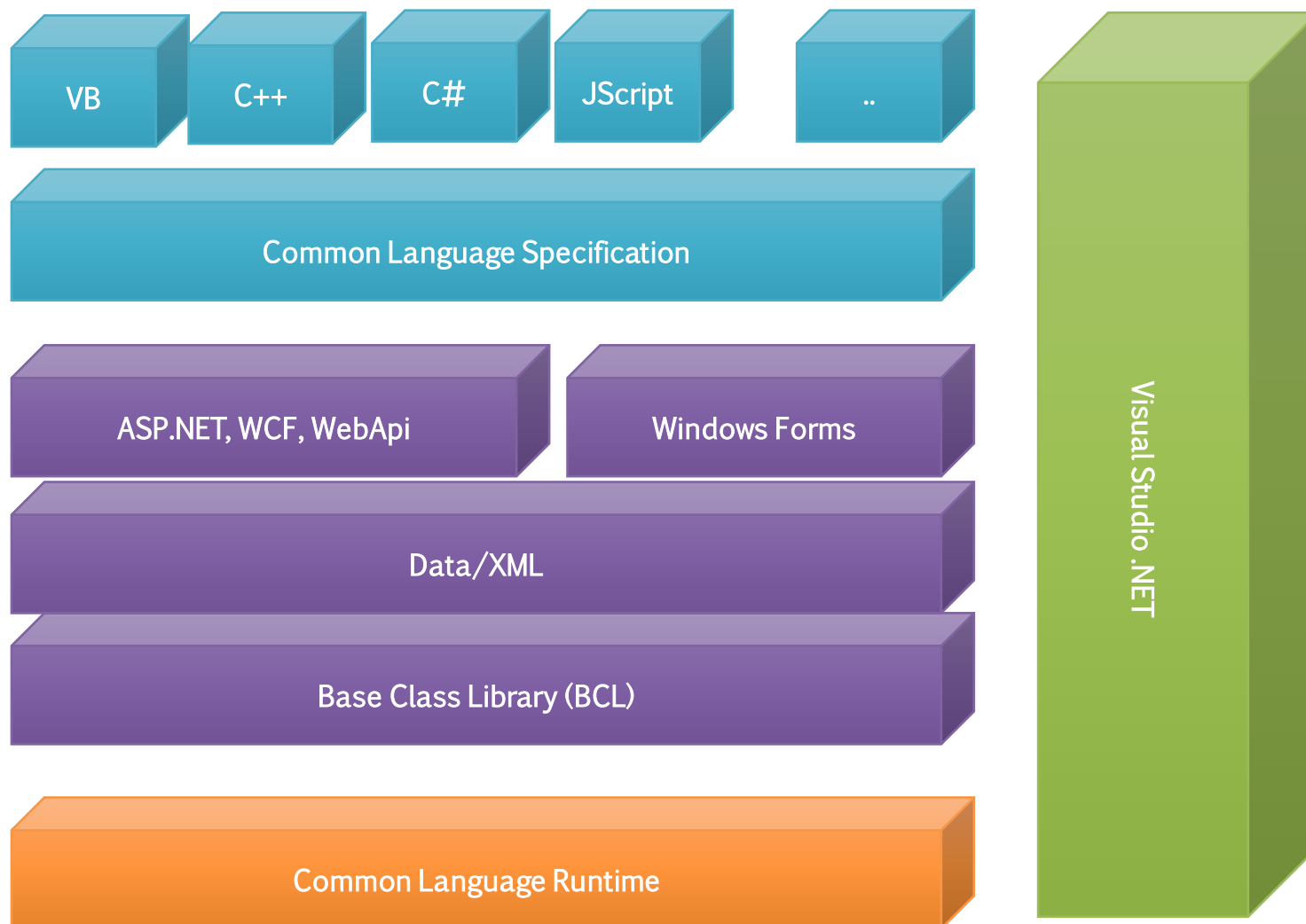
C#

- › The main reference language for programming .NET
- › Object-oriented
- › Portable
- › Managed (???)





The Microsoft .NET architecture





See sharp?

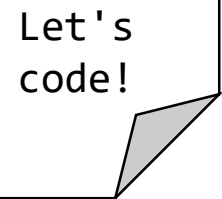




Some background: lambda functions

```
(input-parameters) => expression
```

- › Expression is a portion of code
- › input-parameters can be empty
- › Functions treated as data



Let's
code!



.NET Task model

- › Single program, single task in `Main`
- › Shared memory
- › Then, create parallelism with
 - `Parallel.ForEach` (data decomposition)
 - `Task.Run` construct (tasking)
- › Use lambda functions for parallel code
- › Concept of `Actions` & `Func` (action with return value)
- › Object-oriented language => those constructs work on objects!



Data decomposition: Parallel.ForEach

```
//  
// Summary:  
//     Executes a foreach (For Each in Visual Basic) operation on an  
//     System.Collections.IEnumerable in which iterations may run in parallel.  
//  
// Parameters:  
//     source:  
//     An enumerable data source.  
//  
//     body:  
//     The delegate that is invoked once per iteration.  
//  
// Returns:  
//     A structure that contains information about which portion of the loop  
//     completed.  
  
public static ParallelLoopResult ForEach<TSource>(  
    IEnumerable<TSource> source, Action<TSource> body);
```

- › Executes the same Action on every item of the IEnumerable
 - Action class to represent a portion of code with no return value (for this, use Functions)
 - Typically, implemented with a lambda function
 - IEnumerable common to collection/list/dictionaries/arrays...



Exercise

Let's
code!

Create a list, and iterate over it using `Parallel.ForEach`

- › Lambda function to specify the work to be done

```
Parallel.ForEach (list, (item) => { /* WORK */ })
```

- › Increment a shared counter

- We need to lock!

- › Locks work on objects..

- Can we do better?



Critical and atomicity: Interlock class

```
//  
// Summary:  
//     Provides atomic operations for variables that are shared  
//     by multiple threads.  
public static class Interlocked  
{  
  
    //  
    // Summary:  
    //     Increments a specified variable and stores the result,  
    //     as an atomic operation.  
    public static int Increment(ref int location);  
  
}
```

- › Provides thread-safe operations on variables
 - By means of static functions



Exercise

Let's
code!

- › Create an array of integers

```
int []A = new int[size];
```

- › Multiply item by two

- What do you see?
- Why?

- › We Box integer inside a BoxedInteger object

- Now, we can modify the Counter field of the object
- We can also use the object in lock



Tasking: the Task.Run method

```
//  
// Summary:  
//     Queues the specified work to run on the thread pool and returns a //  
System.Threading.Tasks.Task object that represents that work.  
//  
// Parameters:  
//     action:  
//     The work to execute asynchronously  
//  
// Returns:  
//     A task that represents the work queued to execute in the ThreadPool.  
//  
// Exceptions:  
//     T:System.ArgumentNullException:  
//     The action parameter was null.  
  
public static Task Run(Action action);
```

- › Abstracts the concept of Task delivered to a Thread
 - The name of function can be misleading
- › Uses lambda functions to specify portions of code to execute
- › Returns a Task object



The Task.Run method

docs.microsoft.com

"It's important to reason about tasks as abstractions of work happening asynchronously, and not an abstraction over threading.

By default, tasks execute on the current thread and delegate work to the Operating System, as appropriate.

Optionally, tasks can be explicitly requested to run on a separate thread via the Task.Run API."



The Task.Wait method

```
//  
// Summary:  
//     Waits for the System.Threading.Tasks.Task to complete execution.  
//  
// Exceptions:  
//     T:System.ObjectDisposedException:  
//         The System.Threading.Tasks.Task has been disposed.  
//  
//     T:System.AggregateException:  
//         The task was canceled. The  
//         System.AggregateException.InnerExceptions collection contains a  
//         System.Threading.Tasks.TaskCanceledException object.  
//         -or- An exception was thrown during the execution of the task.  
//         The System.AggregateException.InnerExceptions  
//         collection contains information about the exception or exceptions.  
  
public void Wait();
```

- › Non-static method
- › The Task object represents the «hook» to manage the async work



Exercise

Let's
code!

› Create a Task

- Use `Thread.Sleep` to simulate computation time
- Do parallel work on the main (implicit) Task, then Wait for the parallel task



Exercise

Let's
code!

- › Create a Task that fetches a webpage

```
new System.Net.Http.HttpClient().GetStringAsync(url);
```

- › What is the return value of GetStringAsync?
- › Can we do better?



The async/await parallel pattern

[eng.wikipedia.org](https://en.wikipedia.org)

In computer programming, the `async/await` pattern is a syntactic feature of many programming languages that allows an asynchronous, non-blocking function to be structured in a way similar to an ordinary synchronous function.

How to use it in C#

- › Use `await` keyword when calling functions that return a `Task`
- › Use `async` keywords for functions that have `await` inside
- › Can be “chained”!!!

C# Tasks implement the concept of promise

- › an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is not yet complete.



Exercise

Let's
code!

- › Create a Task that fetches a webpage

```
new System.Net.Http.HttpClient().GetStringAsync(url);
```

Now, use `async/await`

- › What is the return value of `GetStringAsync`?
- › What is the type of the variable assigned from `GetStringAsync`?
- › Chain them!



Exercise

Let's
code!

Implement a WebServer

- › File -> New Project -> ASP.NET WebApplication (.NET Framework)
 - (.NET Core as minimal subset of the Fwk)
- › Fetch & return a WebPage
 - (Can include the other C# project)
- › Do it both synchronously and asynchronously

Only for Visual Studio atm...sorry..



How to run the examples

Let's
code!

- › Download the Code/ folder from the course website

- › Install Visual Studio (Win) or VS Code (Win & GNU/Linux)
 - <https://visualstudio.microsoft.com/>
 - <https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp>

- › VSCode
 - Add AsyncTasks folder in your workspace, then press F5
 - Might take time, to install all support for C#
 - ATM, cannot run the AsyncWebApp project

- › Visual Studio
 - Open the solution file and choose either of the two as default project, then press F5

References



- › "Calcolo parallelo" website
 - http://hipert.unimore.it/people/paolob/pub/Calcolo_Parallelo/

- › My contacts
 - paolo.burgio@unimore.it
 - <http://hipert.mat.unimore.it/people/paolob/>

- › Useful links
 - <http://www.google.com>
 - <https://en.wikipedia.org/wiki/Async/await>
 - <https://docs.microsoft.com/en-us/dotnet/csharp/async>