# Advanced Development Techniques

## Parallel programming

Sipos Miklós

Óbudai Egyetem Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet
2021

# Contents

# Design aspects

# Calculation of available acceleration

The extent to which the calculation is **faster than the fastest serial execution** code imaginable. (Serial execution time divided by parallel execution time.)

Amdahl's law:

- **Describes the limit of the acceleration reached by parallel execution.**
- **The acceleration is limited by the time of the serial section. (bottleneck)**

It's hard to see where it can be well parallelized and where it's worth it.

If a task results in 1% acceleration in parallel, but writing the code takes drastic extra work ➔ it is not worth it.

# Amdahl’s law more precisely (outlook)

This law describes Gene Amdahl’s observation of the extent to which the speed of a system can be increased at most if it is only partially (i.e., only certain elements) accelerated. In parallel programming, this law is used to predict the expected acceleration of processing with multiple processing units.

Amdahl's law (accelerating effect of accelerating partial calculations):

where

- P: accelerated phase of calculation
- S: degree of acceleration of P

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

Special case: parallelization of calculations

where

- F: serial section of calculation
- N: number of execution units

$$\frac{1}{F + (1-F)/N}$$

# Gustafson’s law (outlook)

- This law, named after John L. Gustafson, states that the processing of **large, easily parallelizable** (consisting of repetitive elements) **datasets** can be effectively parallelized.
- Gustafson's law (accelerating effect of parallelization of calculation):

  where:
  - P: number of execution units
  - S: the degree of acceleration resulting
  - alpha: the non-parallel part of the problem

$$S = (1 - a)P + a = P - a(P - 1)$$

- Contrary to Amdahl's law on fixed-size data sets and problems, Gustafson states that **the size of tasks can be adjusted to the available computing capacity**, and in the same time much larger tasks can be solved than if the size of the tasks is independent of the computing capacity.

# Calculation of available acceleration

Calculation for fence painting:

- 30 min preparement (serial)
- 1 fence section = 1 minute
- 30 min cleaning (serial)

➜ painting 300 fence sections take 360 min if execution is serial

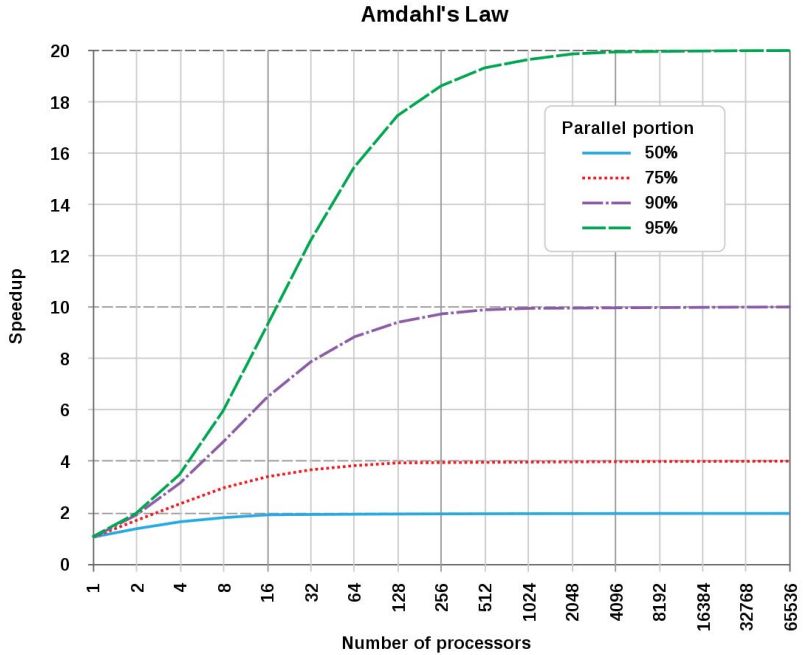
| Number of painters | Time required | Acceleration |
|---|---|---|
| 1 | 30 + 300 + 30 = 360 | 1.0X |
| 2 | 30 + 150 + 30 = 210 | 1.7X |
| 10 | 30 + 30 + 30 = 90 | 4.0X |
| 100 | 30 + 3 + 30 = 63 | 5.7X |
| Infinite | 30 + 0 + 30 = 60 | 6.0X |

The acceleration option is limited by the time of the serial section.

# Calculation of available acceleration

Theoretically, the maximum acceleration that can be achieved is 20 times that of a single thread.

$$\left(\frac{1}{1-p} = 20\right)$$



# Efficiency

The extent to which computing resources (threads) perform a useful activity.

Acceleration / number of threads [%]

It is usually expressed as a percentage in regards of the useful activity time.

| Number of painters | Time required | Acceleration | Efficiency |
|---|---|---|---|
| 1 | 360 | 1.0X | 100% |
| 2 | 30 + 150 + 30 = 210 | 1.7X | 85% |
| 10 | 30 + 30 + 30 = 90 | 4.0X | 40% |
| 100 | 30 + 3 + 30 = 63 | 5.7X | 5.7% |
| Infinite | 30 + 0 + 30 = 60 | 6.0X | Quite bad |

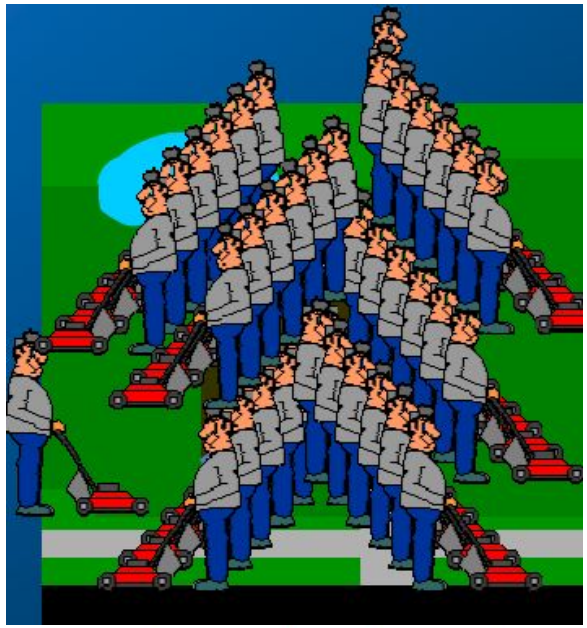
(1.7 / 2) * 100

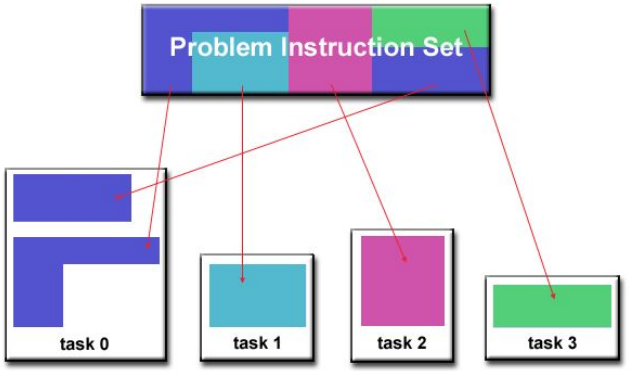
# Granularity

Make sure there is **enough work for each thread**.

Handling too little work and/or having too much threads degrades performance equally.

It is clear that even if I put 1000 lawnmowers on a given unit of area, there will be no acceleration available above a level, in fact.





# Load balancing

Splitting (or partitioning) is most efficient when **each thread** does an **equal amount** of work.

- threads that have already completed tasks are in idle state
  - meanwhile no useful work is done!
- therefore the threads should be finish at the same time

# Additional options for parallel programming

# Additional options for implementing multithreading

- Asynchronous methods
- Async-Await
- ThreadPool
- Task Parallel Library
  - TPL / Task
  - TPL / AsyncAwait
  - TPL / Parallel
- BackgroundWorker

# Asynchronous methods

Asynchronous method: after the task starts, but before it is completed, another task can begin.

„Asynchrony is essential for activities that are potentially blocking (...). Access to a web resource sometimes is slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait.

In an asynchronous process, the application can continue with other work that doesn't depend on the web resource until the potentially blocking task finishes.”

Example: file download

# Asynchronous methods

```
static void Main(string[] args)
{
    WebClient wc = new WebClient();
    wc.DownloadFileCompleted += Wc_DownloadFileCompleted;
    wc.DownloadFileAsync(new Uri("http://users.nik.uni-obuda.hu/prog3/Eloadas/
    WHP_EA_07_Process_Thread.pptx"), "a_prezi.pptx");
    Console.WriteLine("A letöltés elindult...");

    Console.ReadLine();
}

private static void Wc_DownloadFileCompleted(object sender,
System.ComponentModel.AsyncCompletedEventArgs e)
{
    Console.WriteLine("Letöltés vége!");
}
```

```
//eredeti, szekvencialis megoldas:
WebClient wc = new WebClient();
Console.WriteLine("A letöltés elindult...");
wc.DownloadFile(new Uri("http://users.nik.uni-obuda.hu/prog3/Eloadas/WHP_EA_07_Process_Thread.pptx"),
"a_prezi.pptx");
Console.WriteLine("Letöltés vége!");
```

# Asynchronous implementation with event

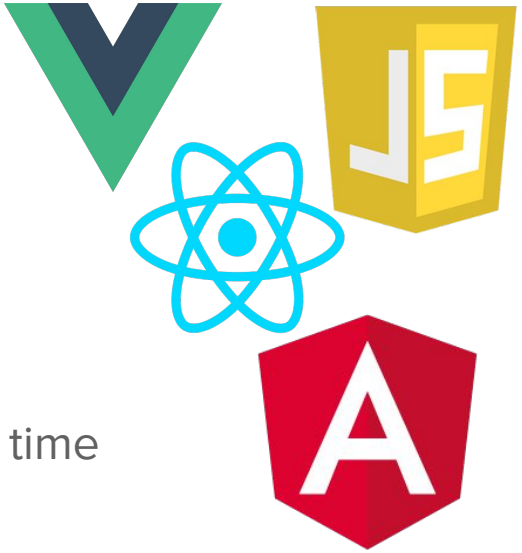
```
static void Main(string[] args)
{
    Console.WriteLine("Megkezdem a letöltést...");
    WebClient wc = new WebClient();
    wc.DownloadStringCompleted += Wc_DownloadStringCompleted;
    wc.DownloadStringAsync(new Uri("http://users.nik.uni-obuda.hu/prog3/_data/war_of_westeros.xml"));

    Console.ReadLine();
}

private static void Wc_DownloadStringCompleted(object sender, DownloadStringCompletedEventArgs e)
{
    Console.WriteLine("Letöltve!");
    Console.WriteLine(e.Result.Contains("Lannister") ? "Van benne Lannister" : "Nincs benne Lannister");
}
```

# Asynchronous implementation with Task

```
Console.WriteLine("Megkezdem a letöltést...");
Task<string> t = new WebClient().DownloadStringTaskAsync("http://users.nik.uni-obuda.hu/prog3/_data/
war_of_westeros.xml");
t.ContinueWith(x => {
    Console.WriteLine("Letöltve!");
    Console.WriteLine(x.Result.Contains("Lannister") ? "Van benne Lannister" : "Nincs benne Lannister");
});
```

Task + ...Async

# Asynchrony in JavaScript (outlook)

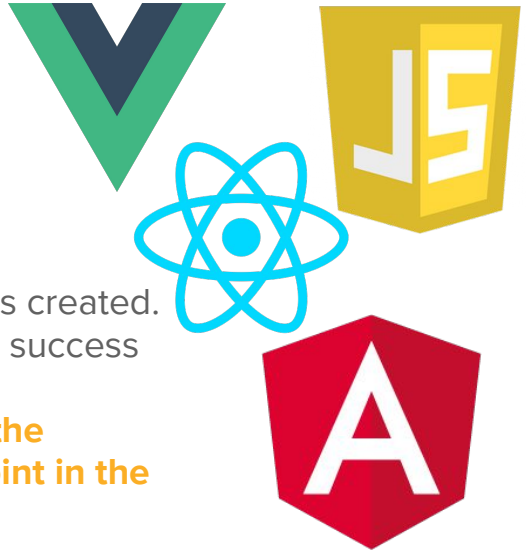


JavaScript is great to code asynchronously:

- The code is asynchronous by default.
- In the code, all function calls which takes longer period of time because eg. you have to wait for a resource, are skipped.
- When the longer process is completed, its return is handled by a so-called callback function.

Keywords: Fetch API, Promise, Event loop.

# Asynchrony in JavaScript (outlook)



- “I promise to do something!”
- A Promise is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: **instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future.** https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- **Used for a long-running task that has a callback.**
- The Fetch API returns back a Promise object
- usage: xxx`.then().then().then()`... ➔ VERY similar to ContinueWith in C#

```
fetch('https://reqres.in/api/users')
    .then(res => res.json()) // console log res
    .then(data => {
        for (let i = 0; i < data.data.length; i++) {
            workerArray.push(data.data[i])
        }
    }).then( () => {
```

# Async-Await

- async
  - Where there is something ‘awaited’, that method must be **marked** with ‘async’ keyword.
    - So the ‘async’ is a method modifier, which marks that inside that method ‘await’ keywords can occur.
  - **Only** void, Task and Task <TResult> return typed methods can be async.
- await
  - A prefix, which marks that **this section should not block the main thread**.
  - Only Tasks can be awaited (with ot without return type).
  - Many built-in language support like GetXXXAsync()

```
async Task<string> AszinkronValasz()
{
    await Task.Delay(2000);
    return "kész";
}
```

# Async-Await (example)

```
static async void Elemez(string url)
{
    Console.WriteLine("Letöltés megkezdve!");
    string tartalom = await new WebClient().DownloadStringTaskAsync(url);
    Console.WriteLine("Letöltve!");
    Console.WriteLine(url + ": " + (tartalom.Contains("Lannister") ? "van benne Lannister" :
"nincs benne Lannister"));
}
```

# Async-Await (example - visual app)

```
async Task<string> AszinkronValasz()
{
    await Task.Delay(2000);
    return "kész";
}

private async void button_Click(object sender, RoutedEventArgs e)
{
    textBox.Text = await AszinkronValasz();
}
```

When you click on a button, something WILL happen but in the meantime you would like to continue your work in the application and maybe clicking on other buttons/menu items etc.

# Async-Await JS (outlook)

Syntactical extra for wrapping promises, to make working easier.

- advantage:
  - no need for nested .then sections
  - no need for chained .then sections
  - more compact, better looking format
- disadvantage
  - a bit of extra work
  - using async and await keywords the corresponding sections must be marked

The same applies to C#!

```
makeRequest('Facebook').then(response => {
  console.log('Response Received')
  return processRequest(response)
}).then(processedResponse => {
  console.log(processedResponse)
}).catch(err => {
  console.log(err)
})
```

```
async function doWork() {
  try {
    const response = await makeRequest('Facebook')
    console.log('Response Received')
    const processedResponse = await processRequest(response)
    console.log(processedResponse)
  } catch (err) {
    console.log(err)
  }
}
doWork()
```

# Async-Await C#

```
class SlowStringMethods
{
    public string GetString()
    {
        Task.Delay(1000).Wait();
        return "wololo";
    }
    public string ReverseString(string str)
    {
        Task.Delay(1000).Wait();
        return new string(str.Reverse().ToArray());
    }
    public string UpperString(string str)
    {
        Task.Delay(1000).Wait();
        return str.ToUpper();
    }
```

```
static void ExecuteBlocking()
{
    Console.WriteLine("EXECUTE BLOCKING");
    SlowStringMethods sm = new SlowStringMethods();
    string str = sm.GetString();
    Console.WriteLine($"RECEIVED: {str}");
    str = sm.ReverseString(str);
    Console.WriteLine($"REVERSED: {str}");
    str = sm.UpperString(str);
    Console.WriteLine($"UPPERCASED: {str}");
}
```

```
Console.WriteLine("EXECUTE with CONTINUATION");
SlowStringMethods sm = new SlowStringMethods();
Task<string>.Factory.
    StartNew(() => sm.GetString()).
    ContinueWith((prevtask) =>
    {
        string str = prevtask.Result;
        Console.WriteLine($"RECEIVED: {str}");
        return sm.ReverseString(str);

    }).ContinueWith((prevtask) =>
    {
        string str = prevtask.Result;
        Console.WriteLine($"REVERSED: {str}");
        return sm.UpperString(str);

    }).ContinueWith((prevtask) =>
    {
        string str = prevtask.Result;
        Console.WriteLine($"UPPERCASED: {str}");
```

```
static async void ExecuteManualAsync()
{
    Console.WriteLine("EXECUTE with MANUAL AWAIT");
    SlowStringMethods sm = new SlowStringMethods();
    string str = await Task<string>.Factory.StartNew(() => sm.GetString());
    Console.WriteLine($"RECEIVED: {str}");
    str = await Task<string>.Factory.StartNew(() => sm.ReverseString(str));
    Console.WriteLine($"REVERSED: {str}");
    str = await Task<string>.Factory.StartNew(() => sm.UpperString(str));
    Console.WriteLine($"UPPERCASED: {str}");
```

# Is Async-Await parallel?

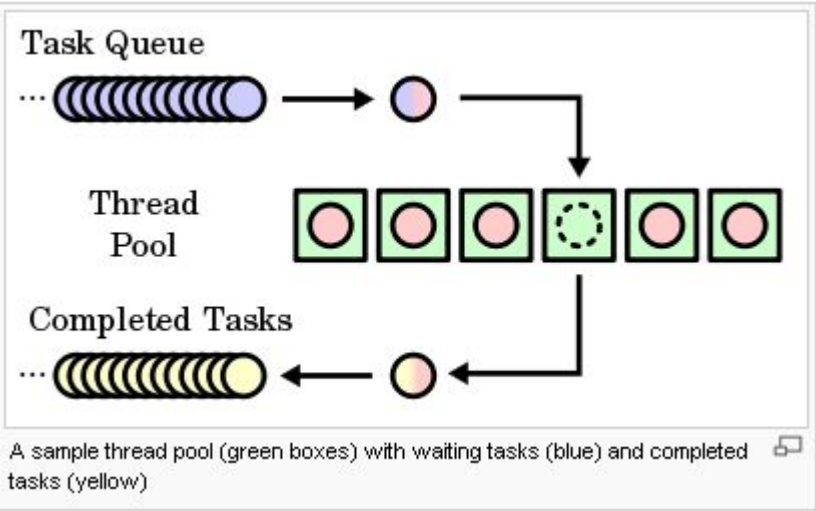
- In fact, there is no separate thread in itself.
- Then how it works?
  - The method marked with the **async** modifier is **executed synchronously until** the call marked with the await keyword.
  - There, however, the method will be in the "suspended" state until the Task completes.
    - The Task, of course, can "choose" to use a separate thread.
  - As long as the method is stopped, the control returns to the call's location and proceeds.
  - When the Task is done, control is given back there.

```
CODE1
await LONG OPERATION WRAPPED IN A TASK
CODE2


CODE1
new Task(LONG OPERATION).ContinueWith(CODE2)
```

# Threadpool

Creating threads (one by one) is expensive, it would be a better idea to register “reusable” threads in the operating system and use them only when needed.

The number of the threads can be different according to the system load.




A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

# Threadpool

- QueueUserWorkItem expects a WaitCallback delegate with a void (object) signature.
- It will be executed when there will be a free pool Thread.

```
static void Main(string[] args)
{
    for (int i = 0; i < 20; i++)
        ThreadPool.QueueUserWorkItem(Kalkulacio, i);

        Console.WriteLine("Várakozunk a nagy válaszra ...");

    Console.ReadLine();
}

static void Kalkulacio(object o)
{
    Thread.Sleep(1000);
    Console.WriteLine(o + ": az élet értelme 42");
}
```

```
Várakozunk a nagy válaszra ...
3: az élet értelme 42
2: az élet értelme 42
1: az élet értelme 42
0: az élet értelme 42
5: az élet értelme 42
4: az élet értelme 42
6: az élet értelme 42
7: az élet értelme 42
8: az élet értelme 42
9: az élet értelme 42
10: az élet értelme 42
12: az élet értelme 42
11: az élet értelme 42
13: az élet értelme 42
14: az élet értelme 42
16: az élet értelme 42
15: az élet értelme 42
18: az élet értelme 42
17: az élet értelme 42
19: az élet értelme 42
```

# TPL

“The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces. **The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications.** The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, **the scheduling of threads on the ThreadPool**, cancellation support, state management, and other low-level details. By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.”

https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl

# TPL

Official examples:

https://docs.microsoft.com/en-us/samples/browse/?products=dotnet-core%2Cdotnet-standard&term=parallel&terms=parallel

Data parallelism:

https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library

PLINQ:

https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq

# Parallel.For

Basically the for loop's body is exposed and an Action<int> delegate is called for every cycle variable.

```
int[] eredm = new int[10];
Parallel.For(0, eredm.Length,
    i => {
        eredm[i] = i * i;
        Console.Write(i + " kész! ");
    }
);

Console.WriteLine();
for (int i = 0; i < eredm.Length; i++)
    Console.Write(i + ": " + eredm[i] + ", ");
```

```
0 kész! 1 kész! 3 kész! 4 kész! 5 kész! 6 kész! 7 kész! 8
kész! 9 kész! 2 kész!
0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64,
9: 81,
```

# Parallel.Invoke

Parallel.Invoke() receives an Action[] input as parameter, which tries to execute delegates in parallel.

The instruction blocks, does not proceed until each Action is not finished.

```
Action<int> valami = x => Console.WriteLine(x * 3);
Parallel.Invoke(
    () => Console.WriteLine("A"),
    () => Console.WriteLine("B"),
    () => Console.WriteLine("C"),
    () => Console.WriteLine("D")//,
    //valami // csak üres bemenetű action-nel működik
);
```

```
Parallel.Invoke(
    BasicAction,          // Param #0 - static method
    () =>                 // Param #1 - lambda expression
    {
        Console.WriteLine("Method=beta,Thread={0}", Thread.CurrentThread.ManagedThreadId);
    },
    delegate ()           // Param #2 - in-line delegate
    {
        Console.WriteLine("Method=gamma,Thread={0}", Thread.CurrentThread.ManagedThreadId);
    }
);
```

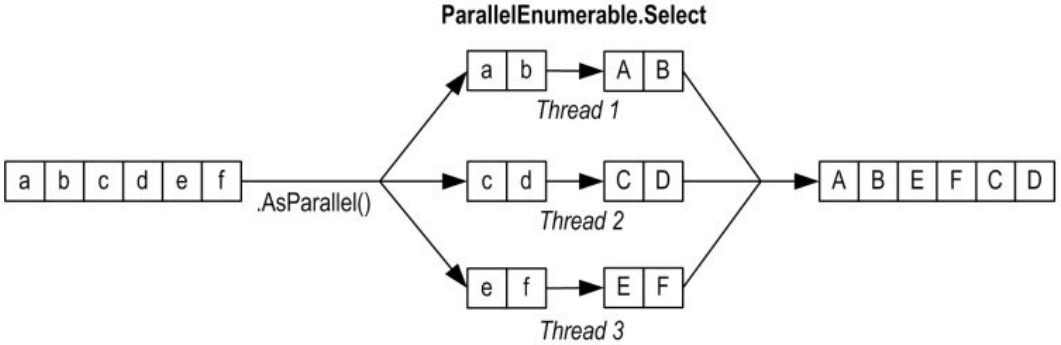
# Parallel.ForEach

Similar to Parallel.For(), uses an Action<T> delegate based on the collection's type (T).

```
Parallel.ForEach("hello world!", e =>
{
    Console.Write(char.ToUpper(e));
});
Console.ReadLine();
```

```
OHELWRD!LO L
```

# Parallel LINQ




```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ToArray()
```

```
// PLINQ

int[] C = new int[] { 2, 4, 6, 8, 3, 1, 5, 7, 2, 0 };

ParallelQuery<int> pq = from x in C.AsParallel()
                        where x % 5 == 0
                        select x;

pq.ForAll(i => DoWork(i));
```

```
➜ /home/sm/Desktop/teszt_plinq dotnet run
Hello World!
2
4
12
6
2344
0
2332

➜ /home/sm/Desktop/teszt_plinq dotnet run
Hello World!
2
2332
0
6
2344
4
12
```

x % 2 == 0

Different runs, different output orders.

```
static void DoWork(int i)
{
    Console.WriteLine(">> " + i);
}
```

# Wrapping slow sections into Tasks

If a class knows that a particular operation may take a long time (e.g., there is an external dependency on an API call, the response may be slow depending on the load on the web server), it is recommended that these methods be handled by Task within the class.

```
new System.Net.WebClient().DownloadString(url);
new System.Net.Http.HttpClient().GetStringAsync(url).Result;
await new System.Net.WebClient().DownloadStringTaskAsync(url);
await new System.Net.Http.HttpClient().GetStringAsync(url);
```

Using bigger C# frameworks this look is quite common!

```
public Task<string> GetStringAsync()
{
    return Task<string>.Factory.StartNew(() => GetString());
}
public Task<string> ReverseStringAsync(string str)
{
    return Task<string>.Factory.StartNew(() => ReverseString(str));
}
public Task<string> UpperStringAsync(string str)
{
    return Task<string>.Factory.StartNew(() => UpperString(str));
}
```

# Sources


Miklós Árpád, Dr. Vámossy Zoltán Design possibilities and methods of parallel algorithms lecture slides

Szabó-Resch Miklós Zsolt és Cseri Orsolya Eszter Advanced Programming lecture slides

Dr. Kertész Gábor Programming of Parallel and Distributed Systems lecture slides

MSDN

Sipos Miklós BPROF SzT specialization's Advanced Development Techniques lecture slides

Sipos Miklós GitHub codes

# Thanks for your attention!

**Sipos Miklós**

sipos.miklos@nik.uni-obuda.hu

https://users.nik.uni-obuda.hu/siposm/