



Curso de Extensão
Tecnologias Microsoft



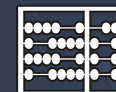
INF0992

Programação Avançada em C#

Aula 2

Hervé Yviquel
hyviquel@unicamp.br

24 de Setembro de 2022



INSTITUTO DE
COMPUTAÇÃO



- Introdução
 - Arquitetura de computadores
 - Paralelismo
 - Desempenho
- Threads
- Tarefas
- Outro paralelismo no C#

The background features a network of gray lines connecting various colored circles (orange, yellow, light blue, light green) and a dark blue horizontal band with a subtle pattern of small gray geometric shapes.

Introdução

Por Que Paralelismo?

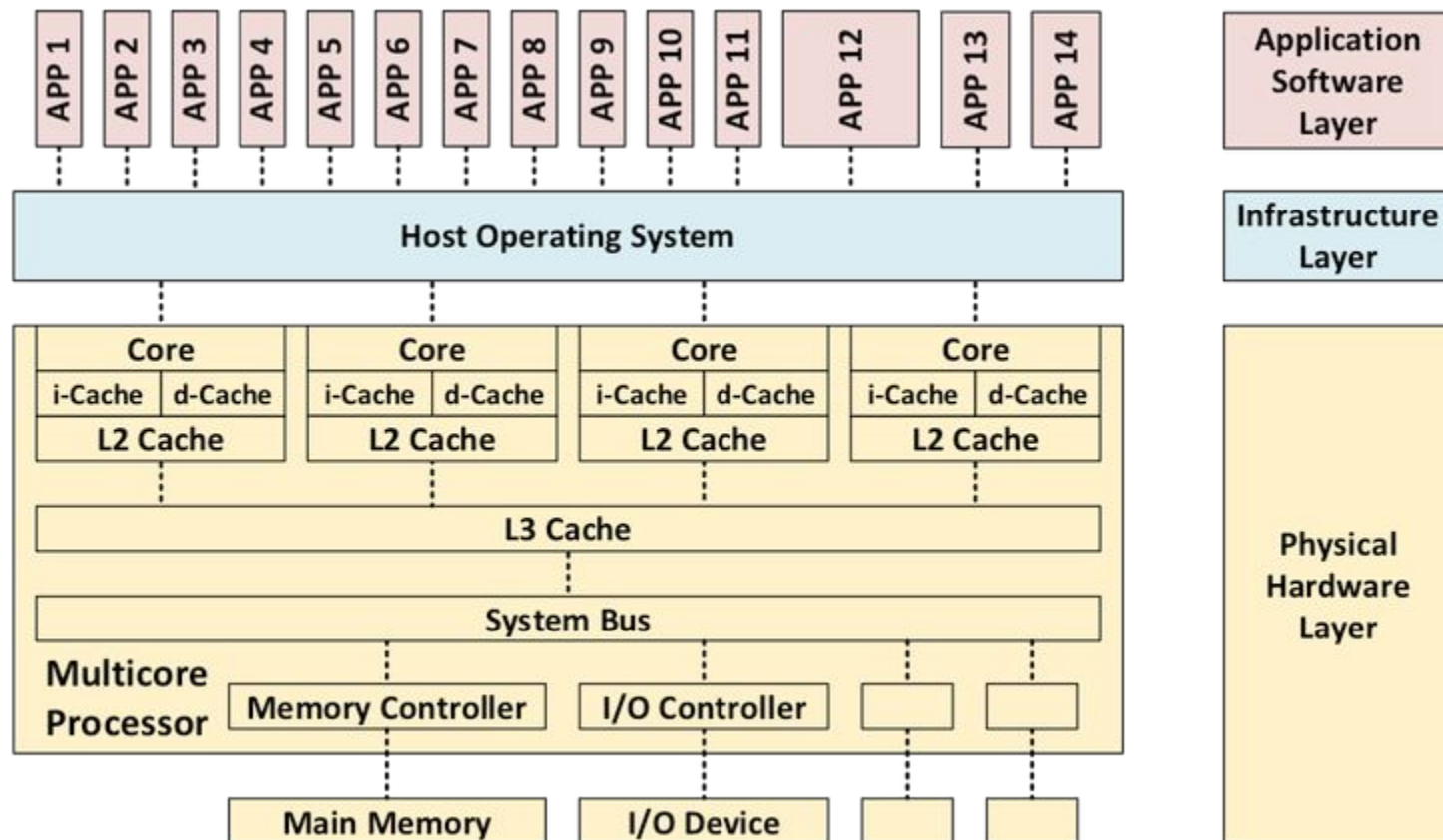


- De 1986 a 2002, o desempenho dos processadores aumentaram muito
 - em média 50% ao ano!
 - Desde então, caiu 20% ao ano
- Até 2002, o aumento de desempenho era devido ao crescimento da densidade dos transistores
- Mas alguns problemas começaram a aparecer....
 - Transistores pequenos = processadores rápidos
 - Processadores rápidos = grande consumo de energia.
 - Grande consumo de energia = aumento na dissipação de calor
 - Grande dissipação de calor = processador não confiável



- Em vez de projetar um processador grande, mais rápido e não confiável com uma grande dissipação de calor
- Use a mesma área de silício para construir muitos processadores menores
 - Chegaram os processadores “multicore” que usamos hoje
 - Cada processador tem vários núcleos que podem fazer computação de forma independente

Arquitetura Multicore



Agora o Trabalho é no Programador!



- Adicionar núcleos não ajuda muito se programadores não estão cientes deles
 - Ou não sabem como usá-los
- Programas sequenciais não se beneficiam disso
 - Na maioria dos casos

Precisamos da
Programação Paralela !!



- **Programação Concorrente**
 - Um programa concorrente é um conjunto de programas sequenciais ordinários os quais são executados em uma abstração de paralelismo
- **Programação Paralela**
 - Consiste em executar simultaneamente várias partes de um mesmo programa
 - O programa é executado pelos varias núcleos do processador

Processos Concorrentes



Task Manager

MININT-N893ED6
Surface Pro 8

CPU 56% GPU 0% Memory 48% Disk 2% Network 2%

Run new task End task View


| Name | Status | 1% CPU | 70% Memory | 1% Disk | 0% Network | Power usage | Power usage fr... |
|------------------------------------|--------|--------|------------|----------|------------|-------------|-------------------|
| Apps (2) | | | | | | | |
| Microsoft Edge (9) | | 0% | 210.6 MB | 0.1 MB/s | 0 Mbps | Very low | Very low |
| Task Manager | | 0.3% | 31.1 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Background processes (34) | | | | | | | |
| AggregatorHost | | 0% | 0.9 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Application Frame Host | | 0% | 4.5 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| COM Surrogate | | 0% | 1.4 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| COM Surrogate | | 0% | 2.7 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| CTF Loader | | 0% | 2.7 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Device Association Framework ... | | 0% | 1.2 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Host Process for Windows Tasks | | 0% | 2.2 MB | 0.1 MB/s | 0 Mbps | Very low | Very low |
| Microsoft OneDrive | | 0% | 4.0 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Microsoft Software Protection P... | | 0% | 3.1 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Microsoft Windows Search Filte... | | 0% | 1.3 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Microsoft Windows Search Inde... | | 0% | 10.1 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Microsoft Windows Search Prot... | | 0% | 2.0 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| Microsoft® Volume Shadow Co... | | 0% | 0.4 MB | 0 MB/s | 0 Mbps | Very low | Very low |
| MoUSO Core Worker Process | | 0% | 3.2 MB | 0 MB/s | 0 Mbps | Very low | Very low |

Settings

Windows 11 Pro Insider Preview
Evaluation copy, Build 22H2.0.rs_prerelease.220114-1500



- Número de núcleos
 - Definido como N
- Tempo de execução serial
 - Definido como T_{serial}
- Tempo de execução paralelo
 - Definido como T_{paralelo}
- Tempo paralelo ideal
 - $T_{\text{paralelo}} = T_{\text{serial}} / N$
 - em geral, o tempo paralelo real é inferior
- Aceleração (ou *speedup*)
 - Definido como $S = T_{\text{serial}} / T_{\text{paralelo}}$

The background features a network of gray lines connecting various colored circles (orange, yellow, blue, green) and a dark blue horizontal band with a white circuit-like pattern.

Threading

Threads – fio/linha/carretel



- Vinculada a um processo
 - Compartilha recursos com o processo
- Possui sua própria pilha de execução
 - Desse ponto de vista, é separado do processo
- Pode coexistir com outras threads do mesmo processo
 - Compartilha dados globais com membros do processo

**Os threads podem
executar em paralelo!**

- 1. Instância o thread**
com o nome da função que
deve executar
- 2. Chama o método start**
para iniciar a execução

13

Funcionamento do Multithreading

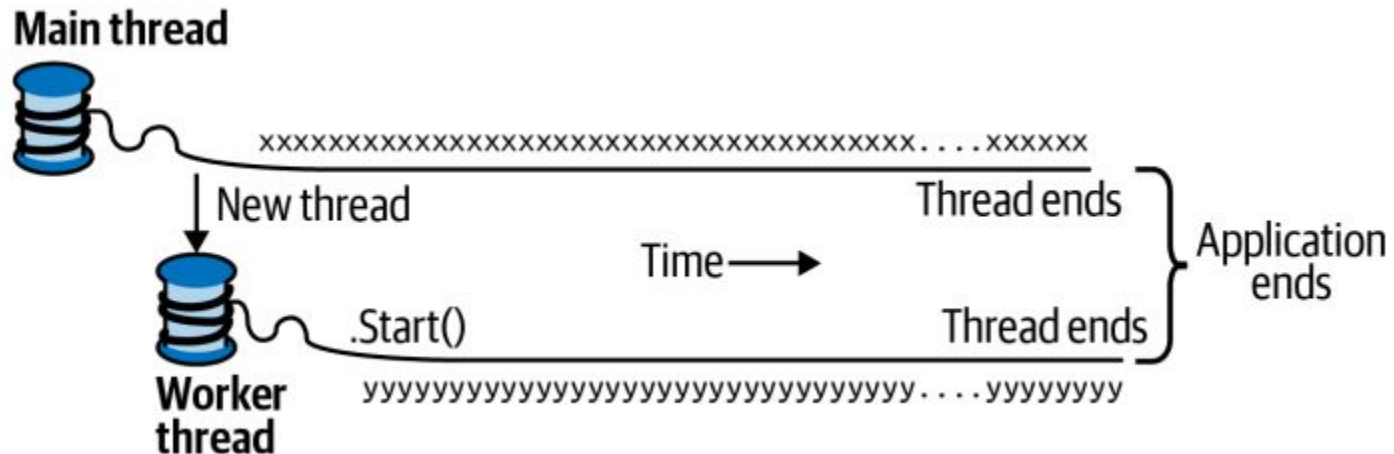


```
using System;
using System.Threading;

Thread t = new Thread (WriteY);           // Kick off a new thread
t.Start();                                 // running WriteY()

// Simultaneously, do something on the main thread.
for (int i = 0; i < 1000; i++) Console.Write ("x");

void WriteY()
{
    for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```



Join e Sleep



- Pode esperar um thread terminar usando *join*
- Pode fazer dormir o thread corrente usando *sleep*
 - durante um determinado período

```
Thread t = new Thread (Go);  
t.Start();  
t.Join();  
Console.WriteLine ("Thread t has ended!");  
  
void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }
```

Printa "y" 1000 vezes
e depois printa que
terminou

```
Thread.Sleep (TimeSpan.FromHours (1)); // Sleep for 1 hour  
Thread.Sleep (500); // Sleep for 500 milliseconds
```


Thread Bloqueado



- Um thread é considerado bloqueado quando sua execução é pausada
 - como a chamada de Sleep
 - ou esperando que outro thread termine por meio de Join
- Um thread bloqueado imediatamente libera o processador
 - não consome mais tempo do processador até que sua condição de bloqueio seja satisfeita

Métodos principais



| Method | Description |
|--------------|--|
| Abort() | is used to terminate the thread. It raises <code>ThreadAbortException</code> . |
| Interrupt() | is used to interrupt a thread which is in <i>WaitSleepJoin</i> state. |
| Join() | is used to block all the calling threads until this thread terminates. |
| ResetAbort() | is used to cancel the Abort request for the current thread. |
| Resume() | is used to resume the suspended thread. It is obsolete. |
| Sleep(Int32) | is used to suspend the current thread for the specified milliseconds. |
| Start() | changes the current state of the thread to <code>Runnable</code> . |
| Suspend() | suspends the current thread if it is not suspended. It is obsolete. |
| Yield() | is used to yield the execution of current thread to another thread. |



- Variáveis locais são privadas aos threads
 - Cada thread tem uma copia própria

```
new Thread (Go).Start();    // Call Go() on a new thread
Go();                      // Call Go() on the main thread

void Go()
{
    // Declare and use a local variable - 'cycles'
    for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

vai imprimir
"?????????"

Variável Compartilhada



- Os threads compartilham as variáveis de escopo maior
 - a memória compartilhada serve para os threads comunicar entre-se
 - mas isto pode gerar problema chamado de “condição de corrida”

```
var tt = new ThreadTest();
new Thread (tt.Go).Start();
tt.Go();

class ThreadTest
{
    bool _done;

    public void Go()
    {
        if (!_done) { _done = true; Console.WriteLine ("Done"); }
    }
}
```

Os 2 threads executam o método do mesmo objeto

É possível que só um ou os 2 threads printam “Done”

**Não
determinismo!!**

Exclusão mútua



- Queremos garantir que um recurso seja acessado apenas por um thread de cada vez
 - pode usar a função lock em qualquer referência

```
class ThreadSafe
{
    static bool _done;
    static readonly object _locker = new object();

    static void Main()
    {
        new Thread (Go).Start();
        Go();
    }

    static void Go()
    {
        lock (_locker)
        {
            if (!_done) { Console.WriteLine ("Done"); _done = true; }
        }
    }
}
```

Se um único thread pode
acessar a seção crítica ao
mesmo tempo

Seção Crítica



- Há duas maneiras de usar argumentos com Threads

```
Thread t = new Thread (Print);  
t.Start ("Hello from t!");  
  
void Print (object messageObj)  
{  
    string message = (string) messageObj;  
    Console.WriteLine (message);  
}
```

Na função *start*
(menos flexível)

Não aconselhado porque
precisa de um cast

```
Thread t = new Thread ( () => Print ("Hello from t!") );  
t.Start();  
  
void Print (string message) => Console.WriteLine (message);
```

Usando funções Lambda

Thread e Funções Lambda



- Cuidado com o escopo das variáveis

```
for (int i = 0; i < 10; i++)  
    new Thread (() => Console.Write (i)).Start();
```

Errado

```
for (int i = 0; i < 10; i++)  
{  
    int temp = i;  
    new Thread (() => Console.Write (temp)).Start();  
}
```

Correto

```
string text = "t1";  
Thread t1 = new Thread ( () => Console.WriteLine (text) );  
  
text = "t2";  
Thread t2 = new Thread ( () => Console.WriteLine (text) );  
  
t1.Start(); t2.Start();
```

Errado

Threads e Exceções



```
try
{
    new Thread (Go).Start();
}
catch (Exception ex)
{
    // We'll never get here!
    Console.WriteLine ("Exception!");
}

void Go() { throw null; } // Throws a NullReferenceException
```

Errado

```
new Thread (Go).Start();

void Go()
{
    try
    {
        ...
        throw null; // The NullReferenceException will get caught below
        ...
    }
    catch (Exception ex)
    {
        // Typically log the exception and/or signal another thread
        // that we've come unstuck
        ...
    }
}
```

Correto



- *Foreground versus Background* Threads
 - O programa fica vivo até todos os threads *foreground* terminar
- Prioridade dos Threads
 - `enum ThreadPriority`
- Multithreading com interface gráfica
 - O main thread cuida da interface e usa outro threads para rodar computação longa
 - Podem usar “Synchronization Contexts” para atualizar interface a partir de outros threads
- Notificação de threads usando sinais
 - Muito prático para esperar alguma coisa sem usar o processar
 - Classe `ManualResetEvent`, Funções `WaitOne/Set`

Limitações dos Threads



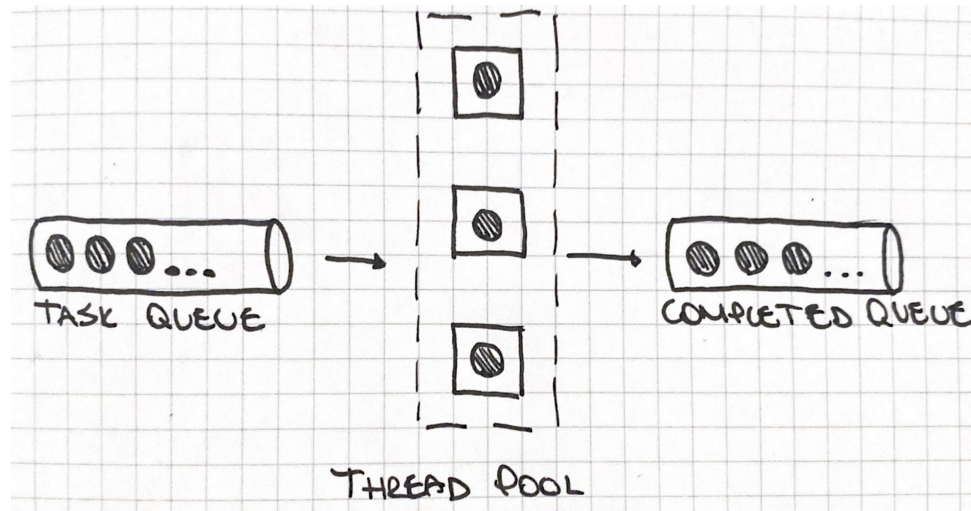
- É uma ferramenta de baixo nível
- Não escala muito bem
 - Criar novos threads tem um custo alto
 - Rodar várias threads no mesmo núcleo pode ter um overhead alto (*context switching*)
- Não é simples de retornar um resultado
 - Poderia usar uma variável global
 - Mas pode passar um argumento facilmente



Tarefas



- Uma tarefa (ou *task*) representa uma unidade de trabalho que deverá ser realizada
 - Tem abstração mais alta que os threads
- Várias *tasks* podem usar o mesmo thread
 - O sistema inicializa um *pool* de threads para executar as tasks



Criação de Tarefa



- Criar tarefa é similar a iniciar um thread

```
Task.Run (() => Console.WriteLine ("Foo"));
```

Task

```
new Thread (() => Console.WriteLine ("Foo")).Start();
```

Thread

Cuidado por que as tarefas
são executadas por threads
do *background* (do pool)

O programa pode
terminar antes que
executa as tarefas

É importante
esperar as
tarefas



- Pode esperar uma task usando o método *wait*
 - Aí vai esperar até a tarefa terminar

```
Task task = Task.Run (() =>
{
    Thread.Sleep (2000);
    Console.WriteLine ("Foo");
});
Console.WriteLine (task.IsCompleted); // False
task.Wait(); // Blocks until task is complete
```

Valor de Retorno da Tarefa



- Tarefas podem retornar valores
 - O valor retornado pode ser acessado usando o campo Result
- Vai bloquear se o valor não for disponível ainda
 - Funcionamento similar ao Future (no Python, C++, Rust, etc)

```
Task<int> task = Task.Run (() => { Console.WriteLine ("Foo"); return 3; });  
// ...
```

```
int result = task.Result;    // Blocks if not already finished  
Console.WriteLine (result); // 3
```

Exemplo:



- Contar os números primos

```
Task<int> primeNumberTask = Task.Run (() =>
    Enumerable.Range (2, 3000000).Count (n =>
        Enumerable.Range (2, (int)Math.Sqrt(n)-1).All (i => n % i > 0)));

Console.WriteLine ("Task running...");
Console.WriteLine ("The answer is " + primeNumberTask.Result);
```

Vai printar "Task running..." e alguns segundos depois



- Tarefas propaga as exceções
 - Ao contrário de thread

```
// Start a Task that throws a NullReferenceException:
Task task = Task.Run (() => { throw null; });
try
{
    task.Wait();
}
catch (AggregateException aex)
{
    if (aex.InnerException is NullReferenceException)
        Console.WriteLine ("Null!");
    else
        throw;
}
```


Funções Assíncronas



- Podem usar Await e Async
 - Funções que usam *await* deve ser marcada como *async*

```
async Task Go()
{
    var task = PrintAnswerToLife();
    await task; Console.WriteLine ("Done");
}
```

```
async Task PrintAnswerToLife()
{
    var task = GetAnswerToLife();
    int answer = await task; Console.WriteLine (answer);
}
```

```
async Task<int> GetAnswerToLife()
{
    var task = Task.Delay (5000);
    await task; int answer = 21 * 2; return answer;
}
```

Conceptualmente *await* é similar ao método *wait* mas não bloqueia o thread

`Task.Delay` é um equivalente assíncrono ao `Thread.sleep`

Paralelismo de Tarefa



- Podemos executar duas funções assíncronas em paralelo
 - esperar depois pelas duas tarefas termina o paralelismo

```
var task1 = PrintAnswerToLife();  
var task2 = PrintAnswerToLife();  
await task1; await task2;
```

*task1 e task2
vão rodar
em paralelo*

```
async Task PrintAnswerToLife()  
{  
    var task = GetAnswerToLife();  
    int answer = await task; Console.WriteLine (answer);  
}  
  
async Task<int> GetAnswerToLife()  
{  
    var task = Task.Delay (5000);  
    await task; int answer = 21 * 2; return answer;  
}
```

WhenAny e WhenAll



- Funções *WhenAny* e *WhenAll* são práticas para esperar a execução de tarefas

```
async Task<int> Delay1() { await Task.Delay (1000); return 1; }  
async Task<int> Delay2() { await Task.Delay (2000); return 2; }  
async Task<int> Delay3() { await Task.Delay (3000); return 3; }
```

```
Task<int> winningTask = await Task.WhenAny (Delay1(), Delay2(), Delay3());  
Console.WriteLine ("Done");  
Console.WriteLine (winningTask.Result); // 1
```

```
await Task.WhenAll (Delay1(), Delay2(), Delay3());
```

=

```
Task task1 = Delay1(), task2 = Delay2(), task3 = Delay3();  
await task1; await task2; await task3;
```

The background features a network of gray lines connecting various colored circles (orange, yellow, blue, green) and a pattern of small, faint circuit-like icons. A thick green horizontal bar is positioned above the title, and another is below it.

Outro Paralelismo no C#




- **Parallel.Invoke**
 - Executa um array de delegados em paralelo
- **Parallel.For**
 - Executa o equivalente paralelo de um loop for
- **Parallel.ForEach**
 - Executa o equivalente paralelo de um loop foreach

```
Parallel.Invoke (  
    () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),  
    () => new WebClient().DownloadFile ("http://microsoft.com", "ms.html"));
```



- Executa um array de ações
 - e as espera terminar

```
public static void Invoke (params Action[] actions);
```



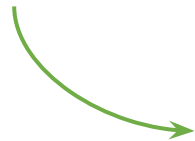
```
Parallel.Invoke (  
    () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),  
    () => new WebClient().DownloadFile ("http://microsoft.com", "ms.html"));
```

Vai baixar as
duas páginas
em paralelo

Laços Paralelos




```
for (int i = 0; i < 100; i++)  
    Foo (i);
```



```
Parallel.For (0, 100, i => Foo (i));
```

```
foreach (char c in "Hello, world")  
    Foo (c);
```



```
Parallel.ForEach ("Hello, world", Foo);
```

LINQ em Paralelo (PLINQ)



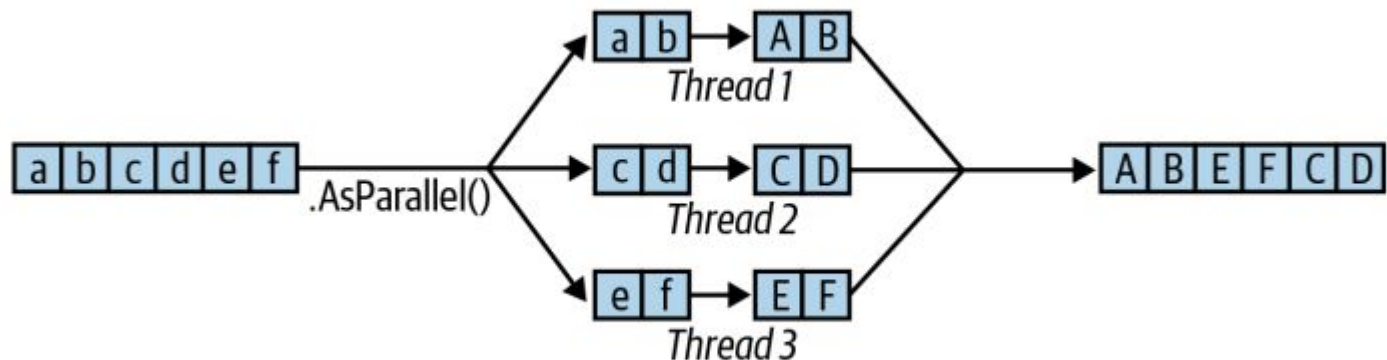
```
// Calculate prime numbers using a simple (unoptimized) algorithm.
```

```
IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);
```

```
var parallelQuery =  
    from n in numbers.AsParallel()  
    where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)  
    select n;
```

```
int[] primes = parallelQuery.ToArray();
```

ParallelEnumerable.Select



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


Limitações do PLINQ



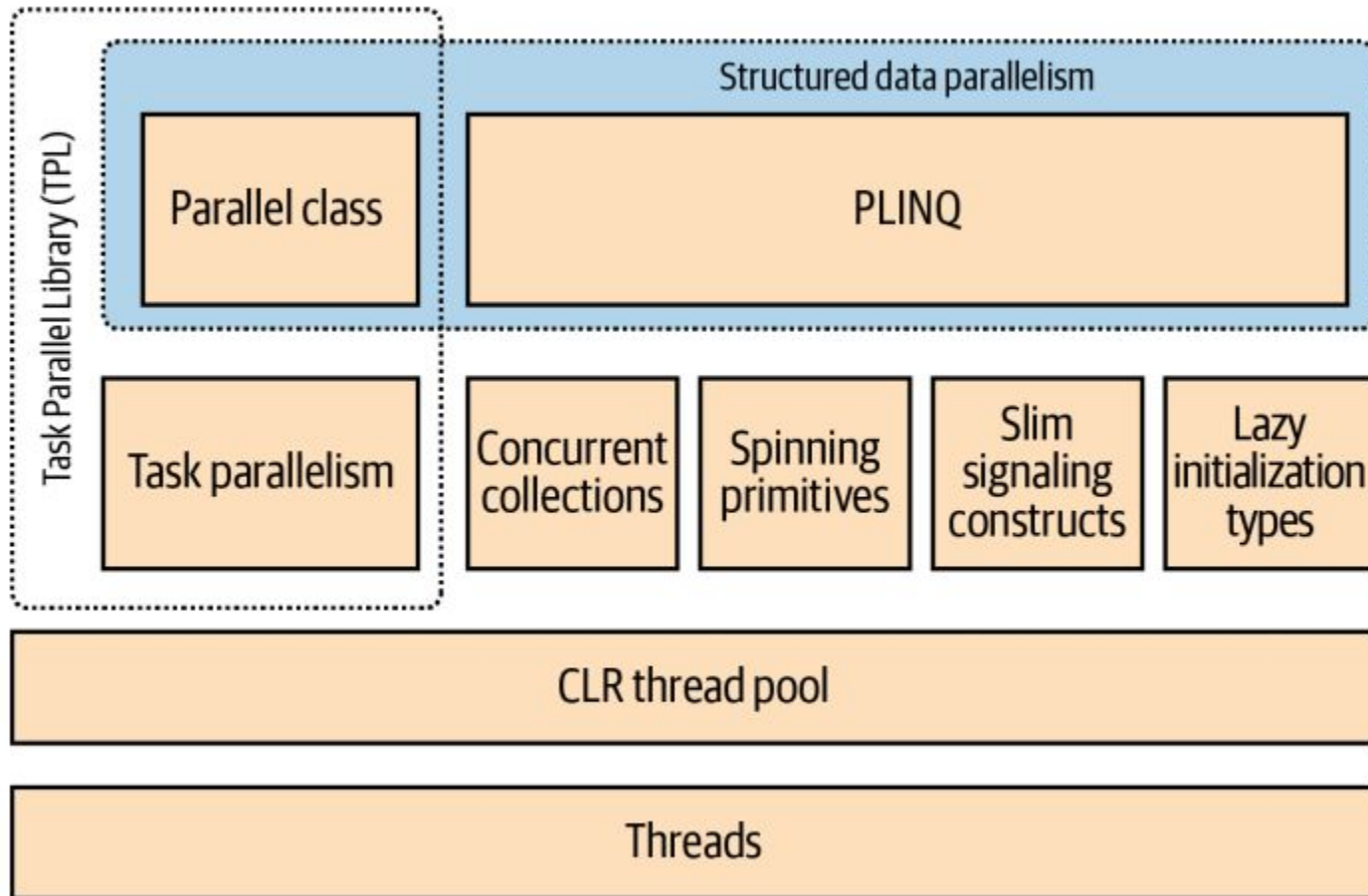
- Ordenação
 - `myCollection.AsParallel().AsOrdered()...`
- Não pode paralelizar
 - As versões indexadas do `Select`, `SelectMany`, e `ElementAt`
- Particionamento pode ser caro
 - `Join`, `GroupBy`, `GroupJoin`, `Distinct`, `Union`, `Intersect`, e `Except`



- Importante notar que as classes de coleção clássica não são *tread-safe*
 - Isso significa que pode ter problema se for acessado em paralelo
 - Deve usar *lock* ou coleções concorrentes

| Concurrent collection | Nonconcurrent equivalent |
|--|--|
| <code>ConcurrentStack<T></code> | <code>Stack<T></code> |
| <code>ConcurrentQueue<T></code> | <code>Queue<T></code> |
| <code>ConcurrentBag<T></code> | (none) |
| <code>ConcurrentDictionary<TKey,TValue></code> | <code>Dictionary<TKey,TValue></code> |

Programação Paralela em C#





Curso de Extensão Tecnologias Microsoft

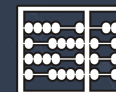


INF0992

Obrigado !!
Merci !!

Hervé Yviquel
hyviquel@unicamp.br

24 de Setembro de 2022



INSTITUTO DE
COMPUTAÇÃO