PARALLEL LOOPS

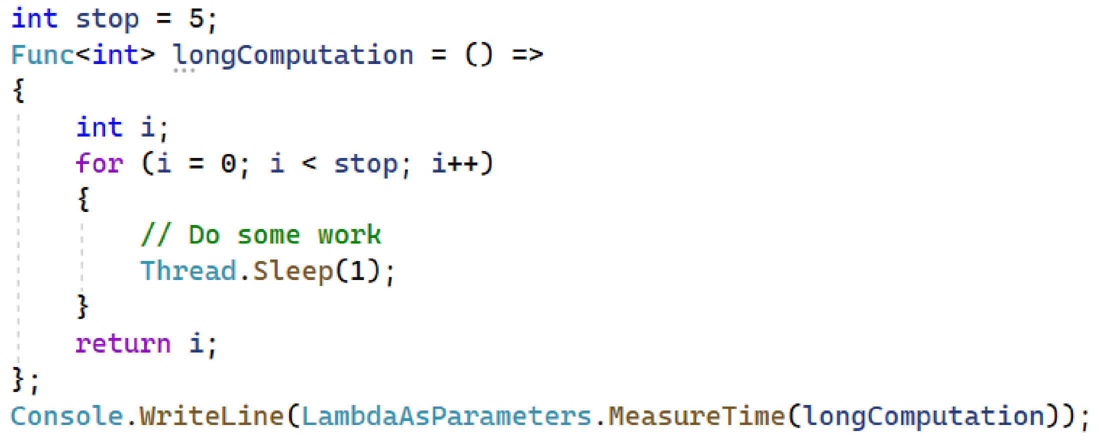
RECAP LAMBDA EXPRESSIONS

```csharp
Func<int, int> square = x => x * x;

Console.WriteLine(square(5));  ──────────→ ?
Console.WriteLine(square);     ──────────→ ?

Action<int> print = i => Console.WriteLine(i);

print(5);
print(10);
```

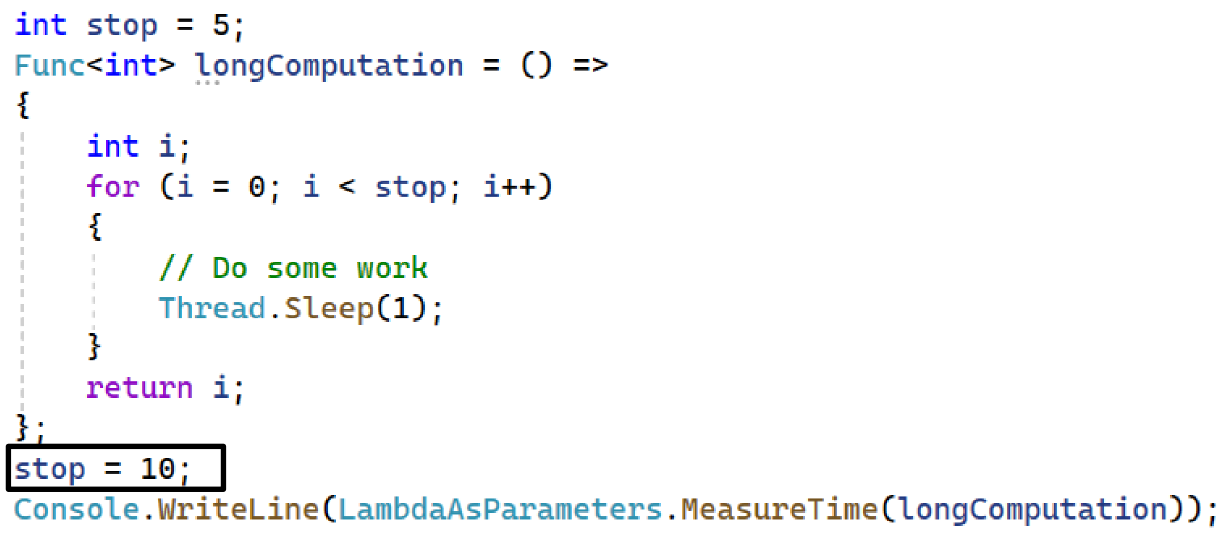
```csharp
Func<int> longComputation = () =>
{
    int i;
    for (i = 0; i < 10; i++)
    {
        // Do some work
        Thread.Sleep(1);
    }
    return i;
};
Console.WriteLine(LambdaAsParameters.MeasureTime(longComputation));
```

```csharp
2 references
internal class LambdaAsParameters
{
    2 references | ✓ 2/2 passing
    public static long MeasureTime(Func<int> someFun)
    {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        Console.WriteLine(someFun());
        stopwatch.Stop();
        return stopwatch.ElapsedMilliseconds;
    }
}
```

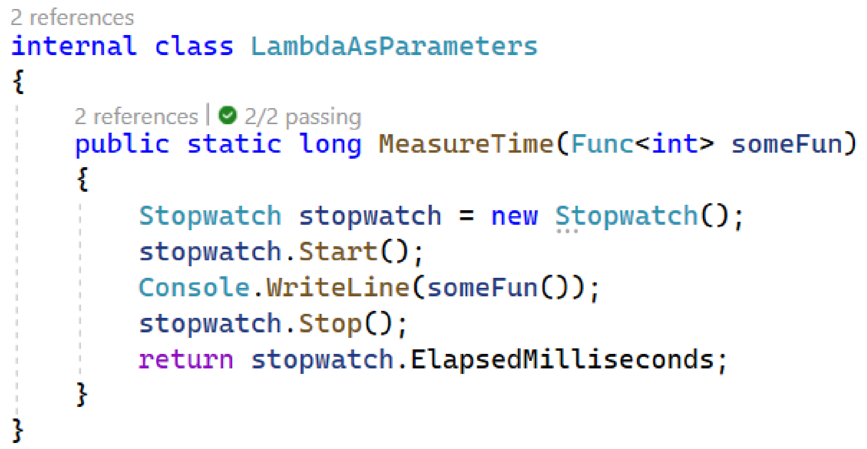
RECAP LAMBDA EXPRESSIONS



```
int stop = 5;
Func<int> longComputation = () =>
{
    int i;
    for (i = 0; i < stop; i++)
    {
        // Do some work
        Thread.Sleep(1);
    }
    return i;
};
Console.WriteLine(LambdaAsParameters.MeasureTime(longComputation));
```



```
int stop = 5;
Func<int> longComputation = () =>
{
    int i;
    for (i = 0; i < stop; i++)
    {
        // Do some work
        Thread.Sleep(1);
    }
    return i;
};
stop = 10;
Console.WriteLine(LambdaAsParameters.MeasureTime(longComputation));
```



```
2 references
internal class LambdaAsParameters
{
    2 references | 2/2 passing
    public static long MeasureTime(Func<int> someFun)
    {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        Console.WriteLine(someFun());
        stopwatch.Stop();
        return stopwatch.ElapsedMilliseconds;
    }
}
```

LOOPING IN APPLICATION

- A significant part of application work is done in loop constructs
- Often, the loop iterations are independent of each other
- When iterations are indeed independent, they may be executed in parallel (PoPP: “delightfully parallel execution”)

```
for(int i=0; i<10; i++)
{
  WriteLine(“i is “ + i)
}
```

$\Longrightarrow$

```
Parallel.For(0, 10, i =>
  {
    Console.WriteLine("i is " + i);
  }
);
```

AN INITIAL IMPLEMENTATION OF PARRALLEL LOOPS

, just to appreciate the problems of partitioning

- TROELS FEDDER

PLANNING IMPLEMENTATION

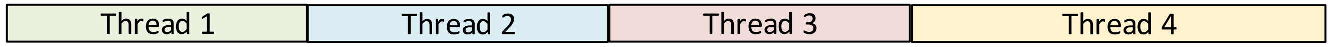
- The parallel loop signature:

```
public static void MyParallelFor(
    int inclusiveLowerBound,
    int exclusiveUpperBound,
    Action<int> body);
```

- Partitioning to individual threads (“1 thread per core”)

```
int size = exclusiveUpperBound - inclusiveLowerBound;
int numProcs = Environment.ProcessorCount;
int range = size / numProcs;
```

Example: `size = 35, numProcs = 4 ⍰ range = 8`

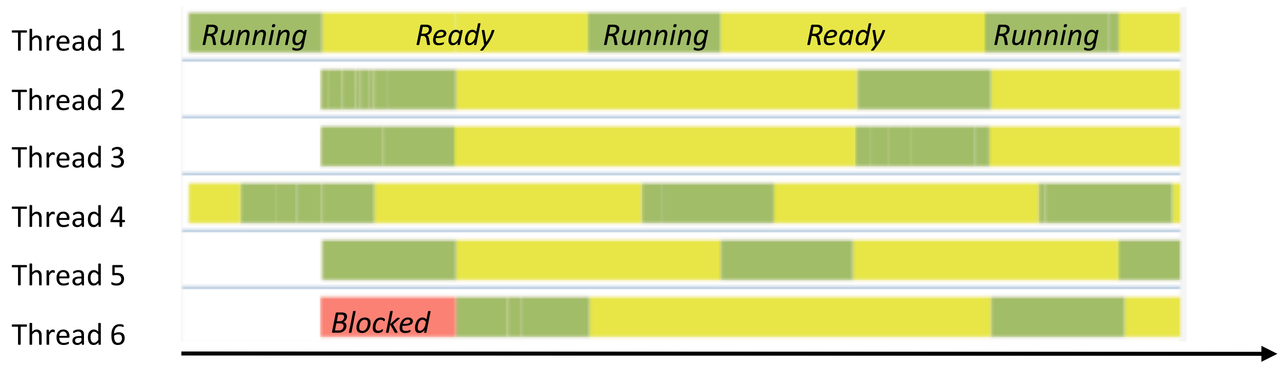


THE IMPLEMENTATION

```csharp
public static void MyParallelFor(int inclusiveLowerBound, int exclusiveUpperBound,
Action<int> body) {
   // Determine size of each partition of work (size/nCores) – static partitioning
   int size = exclusiveUpperBound - inclusiveLowerBound;
   int numProcs = Environment.ProcessorCount;
   int range = size / numProcs;

    // Initialize threads to do work
   var threads = new List<Thread>(numProcs);
   for (int p = 0; p < numProcs; p++)
   {
      int start = p * range + inclusiveLowerBound;
      int end = (p == numProcs - 1) ? exclusiveUpperBound : start + range;
      threads.Add(new Thread(() => {
         for (int i = start; i < end; i++) body(i);
      }));
   }

   // Start and await threads
   foreach (var thread in threads) thread.Start();  // Start them all
   foreach (var thread in threads) thread.Join();   // wait on all
}
```

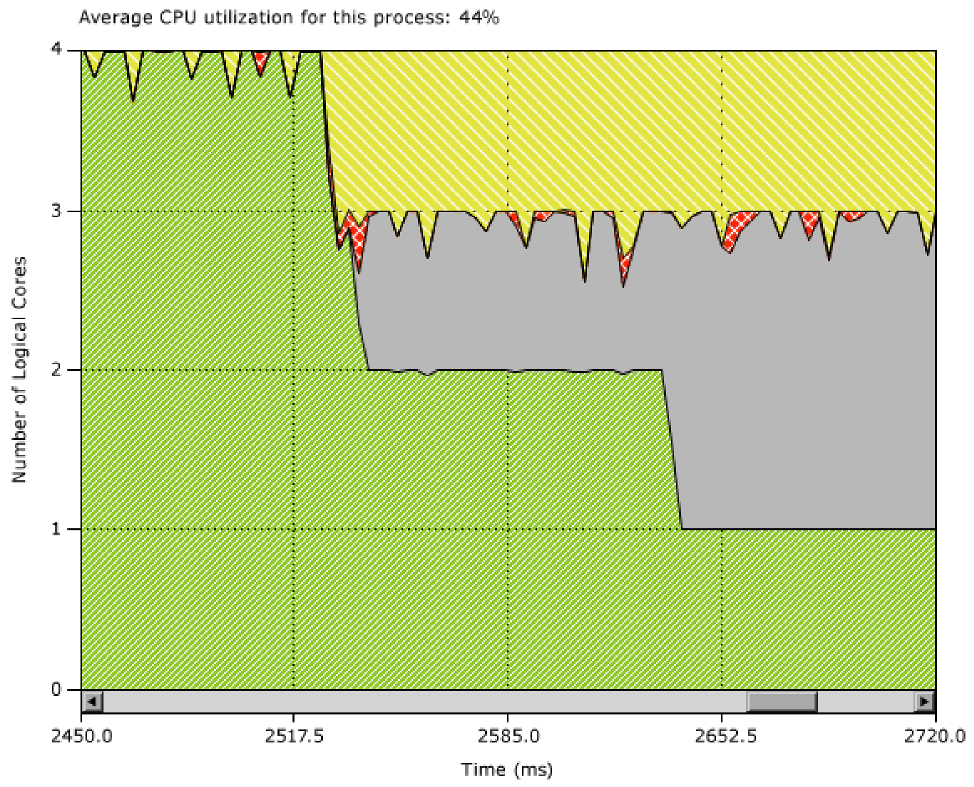
SO ARE ALL GOOD?

- The cost of creating/killing threads is massive!
  - 1 MB stack, 100.000-200.000 cycles for construction/teardown
- The danger of oversubscription:
  - `MyParallelFor()` may itself be called in parallel → 8 (, 12, 16, …) threads for 4 CPUs
  - OS spends time context-switching (takes time, kills caches)
  - Oversubscription example: “yellow is pain!”



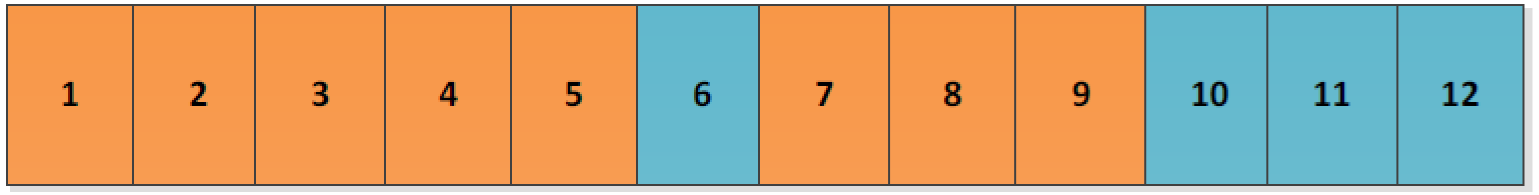
STATIC PARTITIONING

- Static partitioning – load imbalance
  - The in-equivalent workload for each iteration → some threads complete before others
  - Threads represent a static partitioning – threads cannot "help each other out"



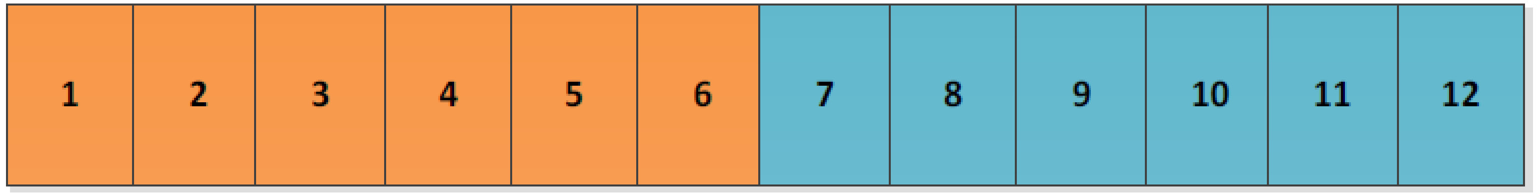
PARTITIONING IS IMPORTANT!

- Assume a parallel loop over `N = [1; 12]`, where iteration i takes i seconds
- Total time to complete = `1+2+3+….+11+12 = 78 secs`
- Ideal load balance (dual core): `78:2 = 39` secs to complete



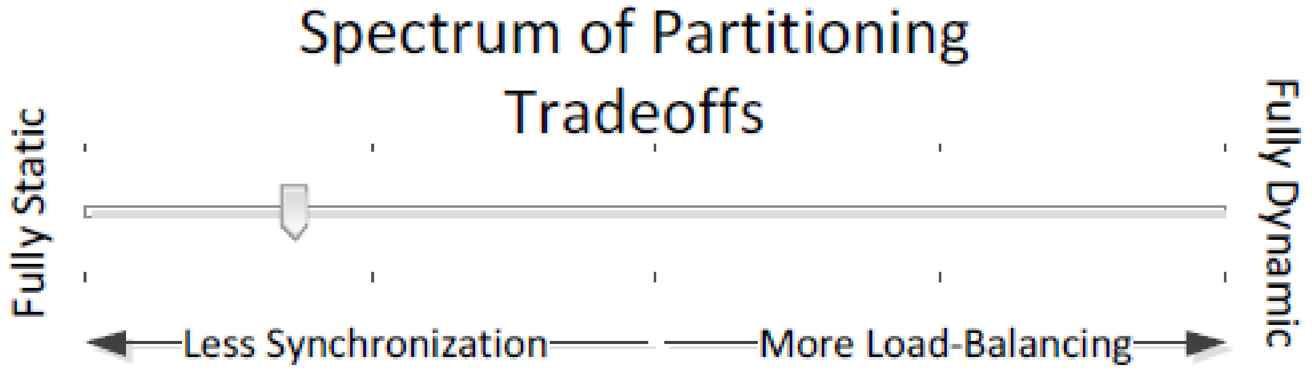
- Our loop uses static load balancing
  - Thread 1: iteration 1 thru 6   21 secs
  - Thread 2: iteration 7 thru 12   57 secs

Show finish order, not proportional time



- Total time to complete loops: 57 secs (46% longer than ideal)

STATIC OR DYNAMIC PARTITIONING



- Effective static partitioning requires a priori knowledge of execution time, but requires no synchronization
  - But is less-than-ideal

- Effective *dynamic* partitioning requires no knowledge but requires synchronization
  - Or the threads will step on each other's toes

ENTER PARALLEL EXTENSIONS

- In .NET 4, the class Parallel was introduced

- Parallel provides 3 static methods (and overloads)
  - `Parallel.For(start, end, Action)`
  - `Parallel.ForEach(collection, Action)`
  - `Parallel.Invoke(Action)`

- `Parallel.For()/ForEach()` provides a number of benefits
  - Exception handling, thread-local state, nested parallelism, dynamic thread count, and sophisticated load balancing

In short: The works!

Parallel.For()

- Parallel version of a regular for-loop:

```
public static ParallelLoopResult For(
   int fromInclusive, int toExclusive, Action<int> body);
```

Example

```
for (int i = 0; i < nCalculations; i++)
  C[i] = Math.Sqrt(Math.Pow(A[i], 2.0) + Math.Pow(B[i], 2.0));
```

Translate to

```
Parallel.For(0, nCalculations, i =>
  {
    C[i] = Math.Sqrt(Math.Pow(A[i], 2.0) + Math.Pow(B[i], 2.0));
  }
);
```

Parallel.ForEach()

- Parallel version of iteration over collection (foreach)

```
public static ParallelLoopResult ForEach<TSource>(
 IEnumerable<TSource> source, Action<TSource> body);
```

Example

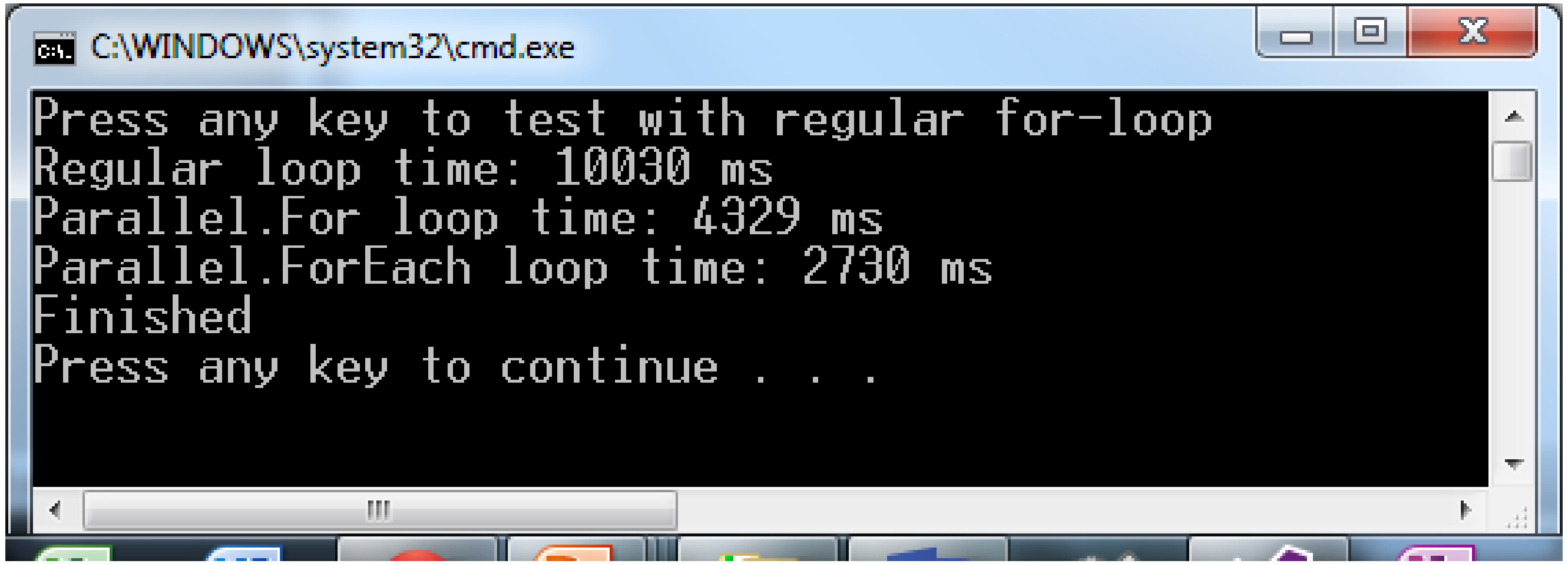
```
foreach (var arg in feArgs) {
  arg.C = Math.Sqrt(Math.Pow(arg.A, 2.0) + Math.Pow(arg.B, 2.0));
}
```

Translate to

```
Parallel.ForEach(feArgs, arg => {
  arg.C = Math.Sqrt(Math.Pow(arg.A, 2.0) + Math.Pow(arg.B, 2.0));
});
```

EXAMPLE: DISTANCE CALCULATIONS

- 1.000.000 coordinates, calculate the distance to each other
- Using regular `for(…)`, `Parallel.For()`, and `Parallel.ForEach()`



```
Press any key to test with regular for-loop
Regular loop time: 10030 ms
Parallel.For loop time: 4329 ms
Parallel.ForEach loop time: 2730 ms
Finished
Press any key to continue . . .
```

AARHUS
UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

THE DANGER ZONES

- To use delightfully parallel loops, the iterations *must* be independent

```
Parallel.For(2, nCalculations, i =>
  {
    a[i] = a[i-1] + a[i-2]; // Oh God, the pain…the PAIN!!
  }
);
```

- Iterations are not always [0..n)
    - Downward iterations `for(..; ..; i--)`
    - Stepped iterations `for(..; ..; I += 2`

- Very small loop bodies may defeat parallelisation
    - Overhead in delegate invocation and load balancing synchronization

OTHER LANGUAGES

- Java
  - `list.parallelStream().forEach()`

  -
    ```
    IntStream.range(0,10)
             .parallel()
             .forEach(i -> ...);
    ```
- C++
  - OpenMP
  - AMP - Accelerated Massive Parallelism
    Using the GPUs on the graphics card.
- Python
  - multiprocessing or joblib

AARHUS
UNIVERSITY