

# IAP C# Lecture 3

## Parallel Computing

Geza Kovacs

# Sequential Execution

- So far, all our code has been executing instructions one after another, on a single core
- Problem: this doesn't make use of the other cores in your system

```
static int sum(int[] list) {  
    int total = 0;  
    foreach (int x in list)  
        total += x;  
    return total;  
}
```

# Parallel Computing

- Splits your task into smaller ones which run simultaneously on multiple cores
- If used properly, will make your code faster
- You can do this a variety of ways in C#:
  - Manipulating threads directly
    - `System.Threading.Thread` class
    - Asynchronous Delegates
  - Task Parallel Library (.NET 4.0)
  - Parallel LINQ (.NET 4.0)
  - Async (CTP available, coming in .NET 5.0)


# Using Threads

- Split your total task into subproblems
- Create and run a thread to solve each subproblem
- Wait for threads to finish, then combine the results
- Optimally, each thread runs on a separate core (but if you have more threads than cores, a single core can swap between threads)

# Creating a Thread

- Thread is defined in namespace System.Threading
- Constructor takes a ThreadStart delegate as an argument

```
public delegate void ThreadStart();
```



No arguments,  
no return value

# Creating a Thread

- Thread is defined in namespace System.Threading
- Constructor takes a ThreadStart delegate as an argument

```
public delegate void ThreadStart();
```

```
int evenSum = 0;  
Thread computeEvens = new Thread(() => {  
    int total = 0;  
    for (int i = 0; i < input.Length; i += 2)  
        total += input[i];  
    evenSum = total;  
});
```



Passing it a lambda expression

# Useful methods for Thread

- `Start()` method starts a thread
- `Join()` method blocks current thread until the thread terminates (returns)
- `Abort()` method forcefully terminates a thread
- `Thread.Yield()` static method yields execution of the current thread (lets other threads execute)
- `Thread.Sleep(int millis)` static method suspends current thread for some time

```
static int sum(int[] list)
{
    int evenSum = 0, oddSum = 0;
    Thread computeEvens = new Thread(() =>
    {
        int total = 0;
        for (int i = 0; i < list.Length; i += 2)
            total += list[i];
        evenSum = total;
    });
    Thread computeOdds = new Thread(() =>
    {
        int total = 0;
        for (int i = 1; i < list.Length; i += 2)
            total += list[i];
        oddSum = total;
    });
    computeEvens.Start(); computeOdds.Start();
    computeEvens.Join(); computeOdds.Join();
    return evenSum + oddSum;
}
```



# Asynchronous Delegates


- Delegate types (like `Func<int, string, bool>`) have `BeginInvoke` and `EndInvoke` methods which allow them to be launched in new threads
- `BeginInvoke`: takes function's arguments, and launches the function in a new thread
  - Has an optional callback, invoked once thread finishes
- `EndInvoke`: blocks until function returns, and gets its return value

```
static int sum(int[] list) {
    Func<int[], int> computeEvens = (input) => {
        int total = 0;
        for (int i = 0; i < input.Length; i += 2)
            total += input[i];
        return total;
    };
    Func<int[], int> computeOdds = (input) => {
        int total = 0;
        for (int i = 1; i < input.Length; i += 2)
            total += input[i];
        return total;
    };
    IAsyncResult evenRes =
        computeEvens.BeginInvoke(list, null, null);
    IAsyncResult oddRes =
        computeOdds.BeginInvoke(list, null, null);
    int evenSum = computeEvens.EndInvoke(evenRes);
    int oddSum = computeOdds.EndInvoke(oddRes);
    return evenSum + oddSum;
}
```

# Task Parallelism

- Often, we create new threads that perform independent tasks, then wait for them to finish executing
  - Ex: one task which sums even numbers, one task which sums odd numbers
- `Parallel.Invoke`, in the `System.Threading.Tasks` namespace, takes a (variable number) of Action delegates, runs them in parallel, and waits for all to return

```
public delegate void Action();
```



No arguments,  
no return value

```
static int sum(int[] list) {
    int evenSum = 0, oddSum = 0;
    Parallel.Invoke(() =>
    {
        int total = 0;
        for (int i = 0; i < list.Length; i += 2)
            total += list[i];
        evenSum = total;
    }, () =>
    {
        int total = 0;
        for (int i = 1; i < list.Length; i += 2)
            total += list[i];
        oddSum = total;
    });
    return evenSum + oddSum;
}
```

# Performance Comparison

- Sequential version: 660 ms

```
using System;
static class MyMainClass {
    static void Main(string[] args) {
        int[] input = new int[100000000];
        for (int i = 0; i < input.Length; ++i)
            input[i] = 1;
        int total = 0;
        var start = DateTime.Now;
        for (int i = 0; i < input.Length; ++i)
            total += input[i];
        var elapsed = DateTime.Now.Subtract(start).TotalMilliseconds;
        Console.WriteLine("milliseconds: " + elapsed);
    }
}
```

# • Threads: 448 ms

```
using System;
using System.Threading;
static class MyMainClass
{
    static int sum(int[] list)
    {
        int evenSum = 0, oddSum = 0;
        Thread computeEvens = new Thread(() =>
        {
            int total = 0;
            for (int i = 0; i < list.Length; i += 2)
                total += list[i];
            evenSum = total;
        });
        Thread computeOdds = new Thread(() =>
        {
            int total = 0;
            for (int i = 1; i < list.Length; i += 2)
                total += list[i];
            oddSum = total;
        });
        computeEvens.Start(); computeOdds.Start();
        computeEvens.Join(); computeOdds.Join();
        return evenSum + oddSum;
    }

    static void Main(string[] args)
    {
        int[] input = new int[100000000];
        for (int i = 0; i < input.Length; ++i) input[i] = 1;
        var start = DateTime.Now;
        Console.WriteLine(sum(input));
        var elapsed = DateTime.Now.Subtract(start).TotalMilliseconds;
        Console.WriteLine("milliseconds: " + elapsed);
    }
}
```

# • Async Delegates: 393 ms

```
using System;
static class MyMainClass
{
    static int sum(int[] list)
    {
        Func<int[], int> computeEvens = (input) =>
        {
            int total = 0;
            for (int i = 0; i < input.Length; i += 2)
                total += input[i];
            return total;
        };
        Func<int[], int> computeOdds = (input) =>
        {
            int total = 0;
            for (int i = 1; i < input.Length; i += 2)
                total += input[i];
            return total;
        };
        IAsyncResult evenRes =
            computeEvens.BeginInvoke(list, null, null);
        IAsyncResult oddRes =
            computeOdds.BeginInvoke(list, null, null);
        int evenSum = computeEvens.EndInvoke(evenRes);
        int oddSum = computeOdds.EndInvoke(oddRes);
        return evenSum + oddSum;
    }

    static void Main(string[] args)
    {
        int[] input = new int[100000000];
        for (int i = 0; i < input.Length; ++i) input[i] = 1;
        var start = DateTime.Now;
        Console.WriteLine(sum(input));
        var elapsed = DateTime.Now.Subtract(start).TotalMilliseconds;
        Console.WriteLine("milliseconds: " + elapsed);
    }
}
```

- Parallel.Invoke: 455 ms

```
using System;
using System.Threading;
using System.Threading.Tasks;
static class MyMainClass
{
    static int sum(int[] list) {
        int evenSum = 0, oddSum = 0;
        Parallel.Invoke(() =>
        {
            int total = 0;
            for (int i = 0; i < list.Length; i += 2)
                total += list[i];
            evenSum = total;
        }, () =>
        {
            int total = 0;
            for (int i = 1; i < list.Length; i += 2)
                total += list[i];
            oddSum = total;
        });
        return evenSum + oddSum;
    }
    static void Main(string[] args)
    {
        int[] input = new int[100000000];
        for (int i = 0; i < input.Length; ++i) input[i] = 1;
        var start = DateTime.Now;
        Console.WriteLine(sum(input));
        var elapsed = DateTime.Now.Subtract(start).TotalMilliseconds;
        Console.WriteLine("milliseconds: " + elapsed);
    }
}
```



# Writes Across Threads

- In example so far, multiple threads read a given variable, but only a single thread writes to it
- What if many threads write to a single variable?

```
using System;
using System.Threading.Tasks;
static class MyMainClass {
    static void Main(string[] args) {
        int counter = 0;
        Parallel.Invoke(() => {
            for (int i = 0; i < 1000000; ++i) {
                ++counter;
            }
        }, () => {
            for (int i = 0; i < 1000000; ++i) {
                --counter;
            }
        });
        Console.WriteLine(counter); // not always 0!
    }
}
```

# Problem with Multi-threaded Counter

- The following sequence of operations might occur:
  - Counter's value is 0
  - Thread a gets counter value (0)
  - Thread b gets counter value (0)
  - Thread a writes back incremented value (1)
  - Thread b writes back decremented value (-1)
- Final counter value is -1!

# Problem with Multi-threaded Counter

- This sequence is also possible (remember, threads interleave operations in unpredictable order)
  - Counter's value is 0
  - Thread a gets counter value (0)
  - Thread b gets counter value (0)
  - Thread b writes back decremented value (-1)
  - Thread a writes back incremented value (1)
- Final counter value is 1!

# Want: Atomic operations

- We want to get and increment, or get and decrement the counter without having it be written to by another thread in the meantime
  - Counter's value is 0
  - Thread a gets counter value (0) and writes back incremented value (1)
  - Thread b gets counter value (1) and writes back decremented value (0)
- Or:
  - Counter's value is 0
  - Thread b gets counter value (0) and writes back decremented value (-1)
  - Thread a gets counter value (-1) and writes back incremented value (0)

- For atomic integer addition, you can use the **Interlocked** static methods (in System.Threading)

```
int counter = 0;
Parallel.Invoke(() =>
{
    for (int i = 0; i < 1000000; ++i)
    {
        Interlocked.Add(ref counter, 1);
    }
}, () =>
{
    for (int i = 0; i < 1000000; ++i)
    {
        Interlocked.Add(ref counter, -1);
    }
});
Console.WriteLine(counter); // always 0
```

- **lock blocks** are a more general way to implement atomic operations. It requires an object instance to be “locked” as the block of code is entered. This prevents other code that needs to lock that object from executing.

```
int counter = 0;
object lockObj = new object();
Parallel.Invoke(() =>
{
    for (int i = 0; i < 1000000; ++i)
    {
        lock(lockObj) { ++counter; }
    }
}, () =>
{
    for (int i = 0; i < 1000000; ++i)
    {
        lock (lockObj) { --counter; }
    }
});
Console.WriteLine(counter); // always 0
```

# Another Example of Parallelism

- Task: return a list of squares of the original input

```
static int[] squareAll(int[] list)
{
    int[] squared = new int[list.Length];
    for (int i = 0; i < list.Length; ++i)
        squared[i] = list[i]*list[i];
    return squared;
}
```



```
static int[] squareAll(int[] list) {
    int[] squared = new int[list.Length];
    Thread computeEvens = new Thread(() => {
        for (int i = 0; i < list.Length; i += 2)
            squared[i] = list[i] * list[i];
    });
    Thread computeOdds = new Thread(() => {
        for (int i = 1; i < list.Length; i += 2)
            squared[i] = list[i] * list[i];
    });
    computeEvens.Start();
    computeOdds.Start();
    computeEvens.Join();
    computeOdds.Join();
    return squared;
}
```

```
static int[] squareAll(int[] list) {
    int[] squared = new int[list.Length];
    Action computeEvens = () => {
        for (int i = 0; i < list.Length; i += 2)
            squared[i] = list[i] * list[i];
    };
    Action computeOdds = () => {
        for (int i = 1; i < list.Length; i += 2)
            squared[i] = list[i] * list[i];
    };
    IAsyncResult evenRes =
        computeEvens.BeginInvoke(null, null);
    IAsyncResult oddRes =
        computeOdds.BeginInvoke(null, null);
    computeEvens.EndInvoke(evenRes);
    computeOdds.EndInvoke(oddRes);
    return squared;
}
```

```
static int[] squareAll(int[] list)
{
    int[] squared = new int[list.Length];
    Parallel.Invoke(() =>
    {
        for (int i = 0; i < list.Length; i += 2)
            squared[i] = list[i] * list[i];
    }, () =>
    {
        for (int i = 1; i < list.Length; i += 2)
            squared[i] = list[i] * list[i];
    });
    return squared;
}
```

# In this problem, Task Parallelism is Suboptimal

- Need to manually split the input among the threads
- Need to have the right number of threads:
  - Too many threads: unnecessary overhead
  - Too few threads: not utilizing all your cores

# Data Parallelism

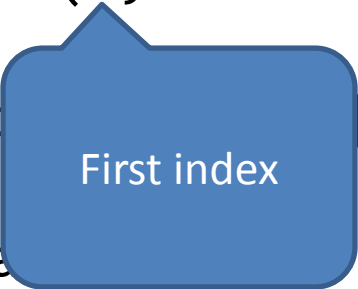
- We are applying the same, single instruction (squaring) to multiple pieces of data
- The result of squaring one element is independent of all other data
- Is there any way to let .NET determine how to best divide up the computations, rather than doing it ourselves?

# Parallel.For

```
static int[] squareAll(int[] list)
{
    int[] squared = new int[list.Length];
    Parallel.For(0, list.Length, (i) =>
    {
        squared[i] = list[i] * list[i];
    });
    return squared;
}
```

# Parallel.For

```
static int[] squareAll(int[] list)
{
    int[] squared = new int[list.Length];
    Parallel.For(0, list.Length, (i) =>
    {
        squared[i] * list[i];
    });
    return squared;
}
```



First index

# Parallel.For

```
static int[] squareAll(int[] list)
{
    int[] squared = new int[list.Length];
    Parallel.For(0, list.Length, (i) =>
    {
        squared[i] = list[i] * list[i];
    });
    return squared;
}
```

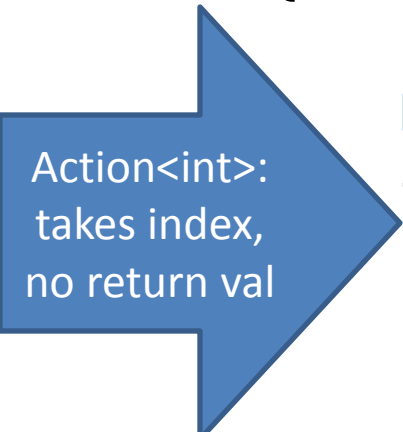


Last index  
(exclusive)



# Parallel.For

```
static int[] squareAll(int[] list)
{
    int[] squared = new int[list.Length];
    Parallel.For(0, list.Length, (i) =>
    {
        squared[i] = list[i] * list[i];
    });
    return squared;
}
```



Action<int>:  
takes index,  
no return val

# Parallel.For

```
static int sum(int[] list)
{
    int total = 0;
    Parallel.For(0, list.Length, (i) =>
    {
        total += list[i];
    });
    return total; // not correct!
}
```

# Parallel.For

```
static int sum(int[] list)
{
    object lockObj = new object();
    int total = 0;
    Parallel.For(0, list.Length, (i) =>
    {
        lock (lockObj) { total += list[i]; }
    });
    return total; // not parallel!
}
```

# Parallel.For

```
static int sum(int[] list)
{
    int numCores = Environment.ProcessorCount;
    object[] locks = new object[numCores];
    for (int i = 0; i < locks.Length; ++i)
        locks[i] = new object();
    int[] sums = new int[numCores];
    Parallel.For(0, list.Length, (i) =>
    {
        lock(locks[i % numCores])
            sums[i % numCores] += list[i];
    });
    int total = 0;
    foreach (int x in sums)
        total += x;
    return total;
}
```

# Parallel LINQ

- Same syntax as LINQ, only add `.AsParallel()`

```
static int[] squared(int[] list)
{
    IEnumerable<int> query = list.Select(x => x * x);
    return query.ToArray();
}
```

```
static int[] squared(int[] list)
{
    IEnumerable<int> query = list.AsParallel().Select(x => x * x);
    return query.ToArray();
}
```

# Parallel LINQ

- Same syntax as LINQ, only add `.AsParallel()`

```
static int[] squared(int[] list)
{
    IEnumerable<int> query =
        from x in list
        select x * x;
    return query.ToArray();
}
```

```
static int[] squared(int[] list)
{
    IEnumerable<int> query =
        from x in list.AsParallel()
        select x * x;
    return query.ToArray();
}
```

```
static int sum(int[] list)
{
    int total = list.Aggregate((runningSum, nextItem) =>
        runningSum + nextItem);
    return total;
}
```

```
static int sum(int[] list)
{
    int total = list.AsParallel()
        .Aggregate((runningSum, nextItem) =>
            runningSum + nextItem);
    return total;
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
static class MyMainClass {
    static int sum(int[] list) {
        int total = list.Aggregate((runningSum, nextItem) =>
            runningSum + nextItem);
        return total;
    }

    static void Main(string[] args) {
        int[] input = new int[100000000];
        for (int i = 0; i < input.Length; ++i)
            input[i] = 1;
        var start = DateTime.Now;
        Console.WriteLine(sum(input));
        var elapsed =
            DateTime.Now.Subtract(start).TotalMilliseconds;
        Console.WriteLine("milliseconds: " + elapsed);
    }
}
```

Without AsParallel: 1700 ms

With AsParallel: 904 ms



```
static int sum(int[] list)
{
    int total = list.Aggregate((runningSum, nextItem) =>
        runningSum + nextItem);
    return total;
}
```

```
static int sum(int[] list)
{
    int total = list.AsParallel()
        .Aggregate((runningSum, nextItem) =>
            runningSum + nextItem);
    return total;
}
```

How does this work? Parallel Reduction.

```
static int sum(int[] list)
{
    int total = list.Sum();
    return total;
}
```

Without AsParallel: 1096 ms

With AsParallel: 546 ms

```
static int sum(int[] list)
{
    int total = list.AsParallel().Sum();
    return total;
}
```