

# Data Parallelism and Control-Flow

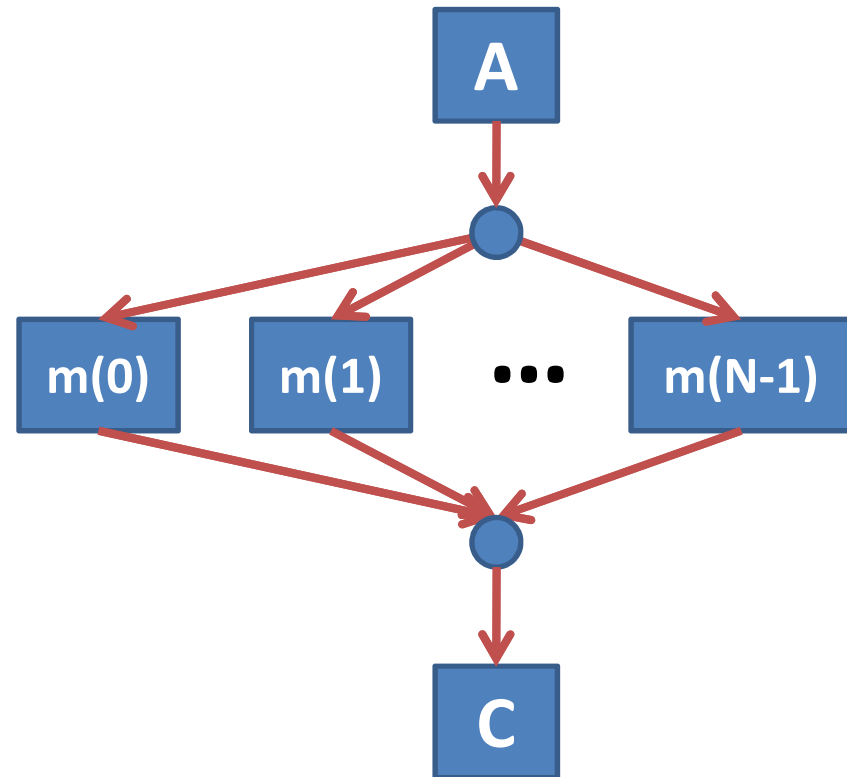
## Unit 1.c

# Acknowledgments

- Authored by
  - Thomas Ball, MSR Redmond

# Recall Parallel.For

```
A;  
Parallel.For(0, N,  
    m: i => { B; }  
);  
C;
```




# Control Flow: When Ordering Matters

- *In theory*, no order between the delegates **m(i)**
  - Parallel.For expresses potential maximal parallelism
- *In practice*, ordering/sequencing of **m(i)** impacts performance
  - cache locality
  - dynamic partitioning, load balancing
- Programmers also may need control over execution
  - stop, break, exception handling, cancellation

# Concepts




Performance  
Concept

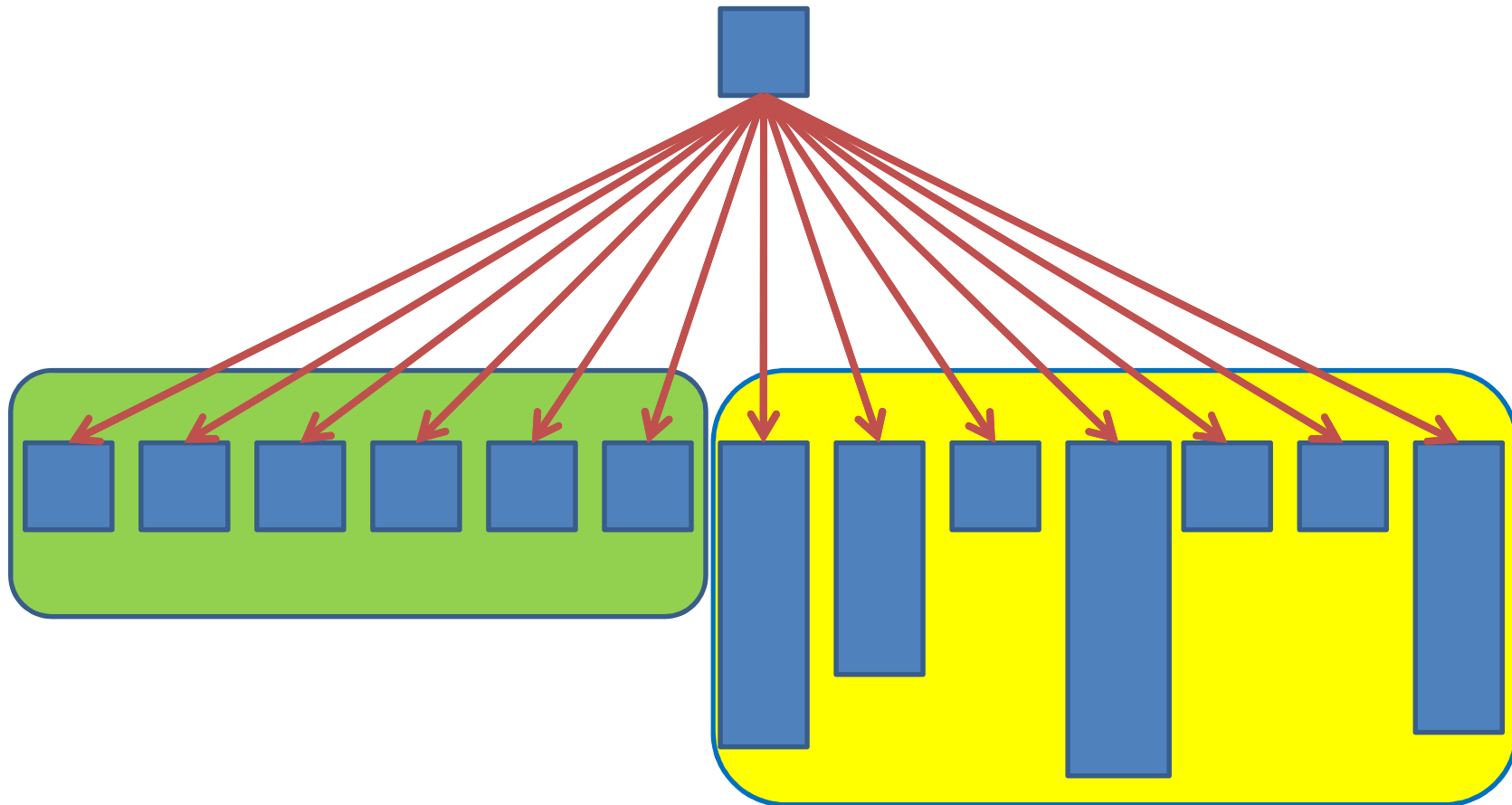
- 
- Workloads
  - Static and dynamic partitioning
  - Coordination vs. computation



Code  
Concept

- 
- Partitioner
  - Local control-flow
  - Stop/break out of Parallel.For
  - Exceptions
  - Cancellation

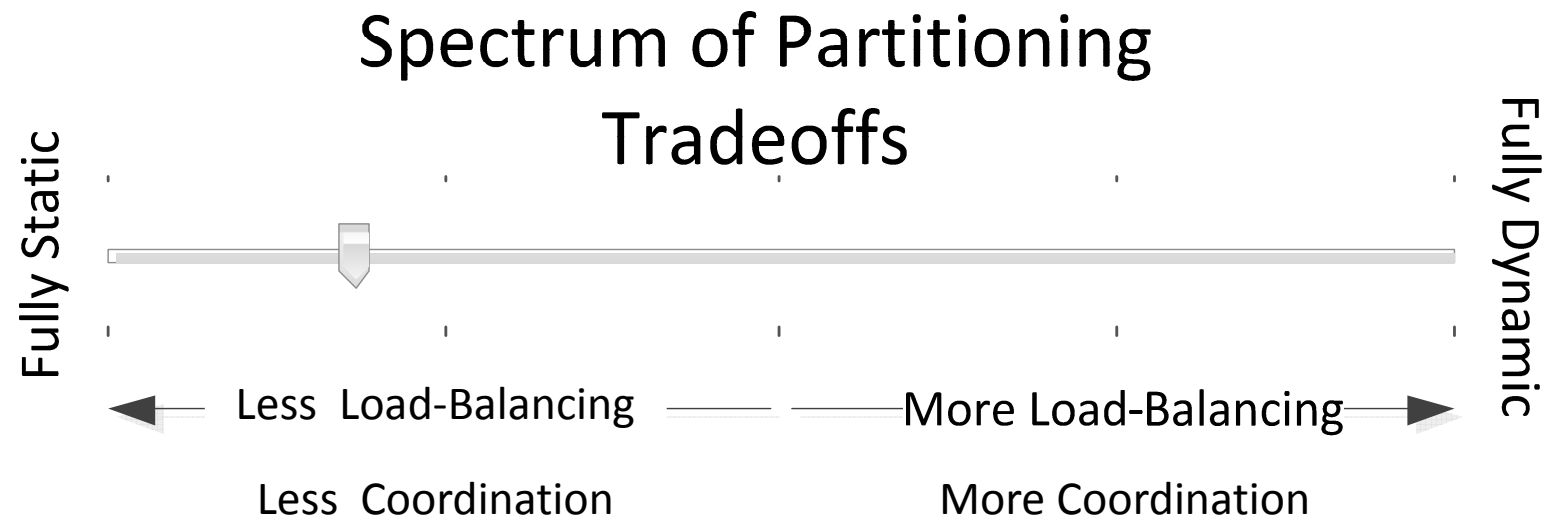
# Parallel.For on Two Cores, an Unbalanced Workload, a Static Partition



# Work and Span

- Parallelism = Work/Span
- Unbalanced workload + static partition
  - Work unchanged
  - Static partition increases Span
  - Parallelism decreases
- What to do?
  - Ensure a balanced workload, or
  - Dynamically partition work (can also increase Span, but not as much as before)

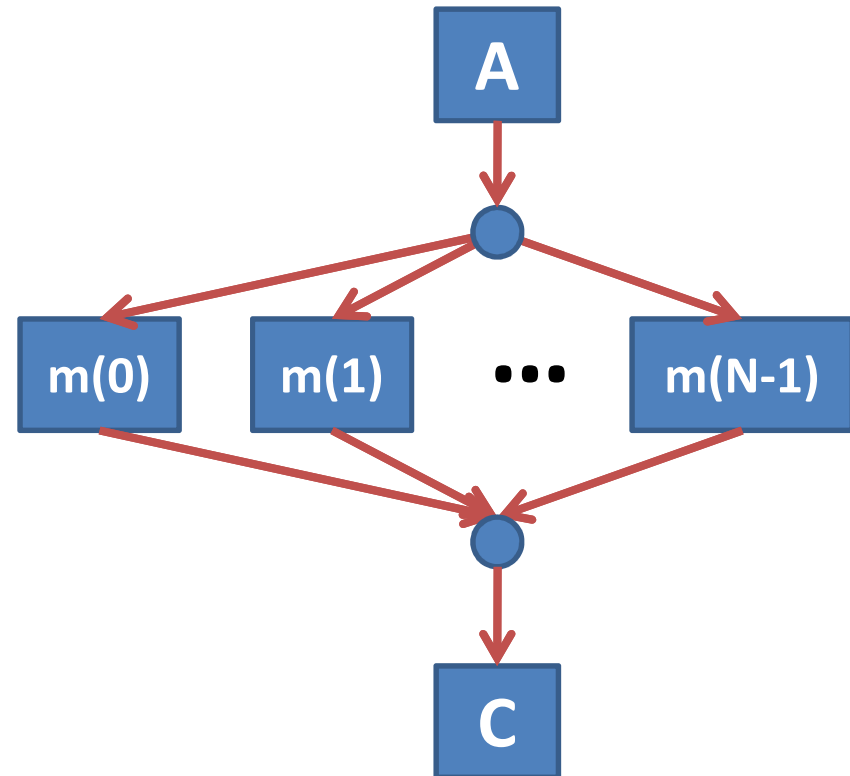
# Partitioning





# Coordination vs. Computation in the Parallel DAG

- Coordination
  - Work done by the run-time to properly schedule  $m(i)$
  - Each edge has a run-time cost (burden)
- Computation
  - The execution of  $A$ ,  $m(i)$ ,  $C$



# Ordering of Iteration Space and Partitioning

- `Parallel.For(0,N, ...)`
  - Ordered by integer range `[0...N-1]`
- `Parallel.ForEach(enumerable, ...)`
  - Ordered by integer range if enumerable is
    - `Array`
    - `IList<T>`
  - Otherwise,
    - ordered by `enumerable.Current/MoveNext`

# Dynamic Partitioning via Chunking

- A *chunk* is a contiguous range of iteration space
  - chunk is executed by one task sequentially
  - more computation/less coordination
- Parallel.For dynamically allocates chunks to tasks
- Chunk size increases over time
  - ensures good load balancing if few iterations
  - minimizes overhead if there are many iterations

# System.Collections.Concurrent.Partitioner

```
// Represents a particular manner of splitting a
// data source into multiple partitions.
public abstract class Partitioner<TSource>
{
    protected Partitioner();
    public virtual bool SupportsDynamicPartitions { get; }
    public virtual IEnumerable<TSource> GetDynamicPartitions();
    public abstract IList<IEnumerator<TSource>>
        GetPartitions(int partitionCount);
}

public abstract class OrderablePartitioner<TSource> :
    Partitioner<TSource>
```



# Take Advantage of Chunking via Local Control Flow

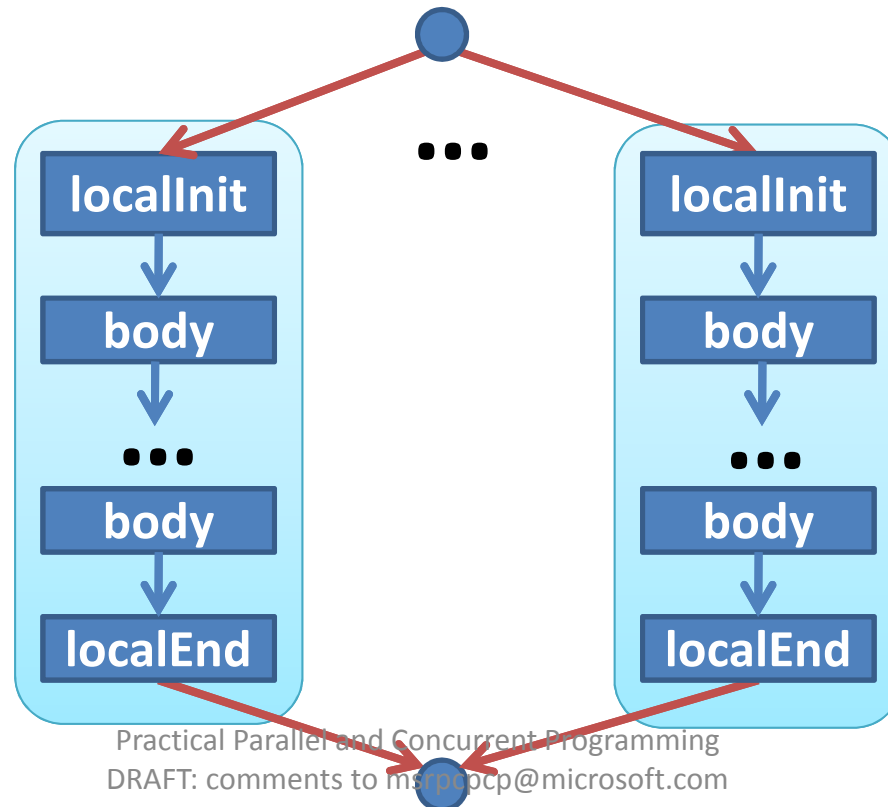
# $\pi$ in Parallel, Very Inefficiently

```
const int NUM_STEPS = 100000000;  
  
static double NaiveParallelPi()  
{  
    double sum = 0.0;  
    double step = 1.0 / (double)NUM_STEPS;  
    object obj = new object();  
    Parallel.For(0, NUM_STEPS, i =>  
    {  
        double x = (i + 0.5) * step;  
        double partial = 4.0 / (1.0 + x * x);  
        lock (obj) sum += partial;  
    });  
    return step * sum;  
}
```

# Local Control-flow

```
public static ParallelLoopResult For<TLocal>(
    int fromInclusive, int toExclusive,
    Func<TLocal> localInit,
    Func<int, ParallelLoopState, TLocal, TLocal> body,
    Action<TLocal> localEnd);
```

Sequential  
Task  
Chunks





# $\pi$ in Parallel, More Efficiently

```
const int NUM_STEPS = 100000000;
static double ParallelPi()
{
    double sum = 0.0;
    double step = 1.0 / (double)NUM_STEPS;
    object obj = new object();
    Parallel.For(0, NUM_STEPS,
        () => 0.0,
        (i, state, partial) =>
        {
            double x = (i + 0.5) * step;
            return partial + 4.0 / (1.0 + x * x);
        },
        partial => { lock (obj) sum += partial; });
    return step * sum;
}
```



ControlFlow\LocalControlFlow.cs

# More Control Flow

- Stop/break out of Parallel.For
- Parallel execution and exceptions
- Cancelling a parallel computation

# Stopping Parallel For Loops

- Example 1
  - Searching a large unsorted collection
  - First hit wins
  
- Example 2
  - Searching a large unsorted collection
  - Find lowest index with matching element

# ParallelLoopState

- Enables iterations of Parallel loops to interact with other iterations
  - One instance provided by runtime to each loop
  - Methods
    - `void Stop()`
    - `void Break()`

# Searching for 42

```
int index = -1;
ParallelLoopResult loopResult =
Parallel.For(0, a.Length,
    (int i, ParallelLoopState loop) =>
    {
        if (a[i] == 42)
        {
            index = i;
            loop.Stop();
        }
    }
);
if (!loopResult.IsCompleted) {
    Console.WriteLine("42 at index " + index);
}
```



ControlFlow\StopBreak.cs

# Searching for First 42

```
ParallelLoopResult loopResult =  
Parallel.For(0, a.Length,  
    (int i, ParallelLoopState loop) =>  
    {  
        if (a[i] == 42)  
        {  
            loop.Break();  
        }  
    }  
);  
if (loopResult.LowestBreakIteration.HasValue) {  
    Console.WriteLine("Lowest index of 42 = " +  
        loopResult.LowestBreakIteration.Value);  
}
```

# Long Running Loop Iterations

- Poll `ParallelLoopState`
  - `bool IsStopped`
  - `Nullable<long> LowestBreakIteration`
  - `bool IsExceptional`
  - `bool ShouldExitCurrentIteration`

# ParallelLoopResult

- **IsCompleted == true**
  - All iterations were processed.
- **IsCompleted == false && LowestBreakIteration.HasValue == false**
  - Stop was used to exit the loop early
- **IsCompleted == false && LowestBreakIteration.HasValue == true**
  - Break was used to exit the loop early,



# What does this code do?

```
try {
    Parallel.Invoke(
        () => { int x = 0; int y = 100/x; },
        () => { object p = null; var s = p.ToString(); }
    );
} catch (Exception e) {
    Console.WriteLine(e.ToString());
}
```

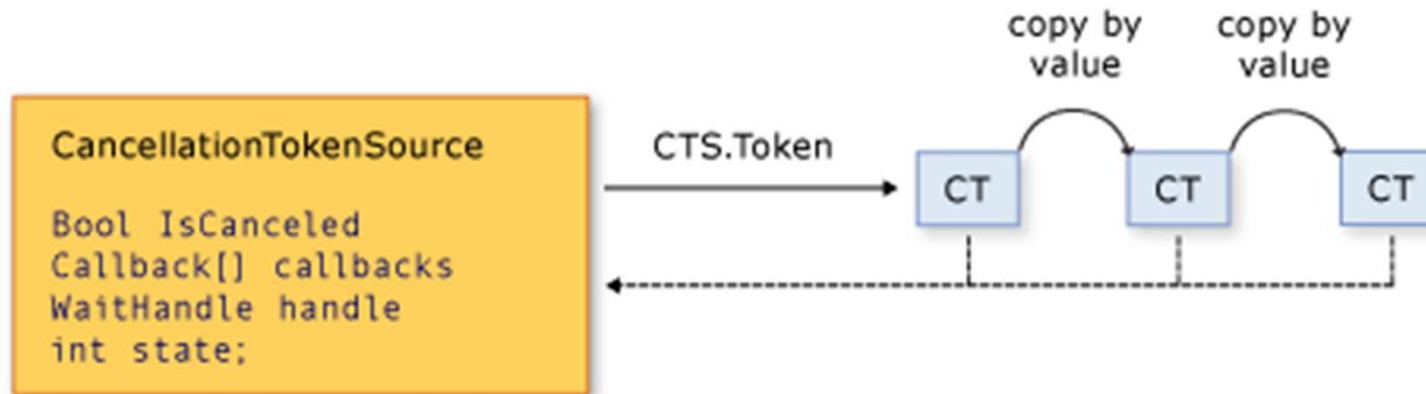
# System.AggregateException

- Used to consolidate multiple failures into a single, throwable exception object
- **AggregateException**
  - `ReadOnlyCollection<Exception> InnerExceptions`
  - `AggregateException Flatten()`
  - `void Handle(Func<Exception, bool> pred)`

ControlFlow\AggregateExceptionExample.cs



# Cancellation in .NET 4



- CancellationTokenSource
- CancellationToken

# Cancellation in .NET 4

- Cancellation is cooperative
- Listeners can be notified of cancellation requests by
  - polling, callback registration, or waiting on wait handles
- A cancellation request is sent to all copies of the token via one method call
- A listener can listen to multiple tokens simultaneously by joining them into one linked token

# void Cancel()

- The associated [CancellationToken](#) will transition to a state where [IsCancellationRequested](#) returns true.
- Any callbacks or cancelable operations registered with the [CancellationToken](#) will be executed.
- Cancelable operations and callbacks registered with the token should not throw exceptions.

# Canceling Parallel.For (1)

```
var cts = new CancellationTokenSource();  
var po = new ParallelOptions()  
{  
    CancellationToken = cts.Token,  
    MaxDegreeOfParallelism =  
        System.Environment.ProcessorCount,  
};
```

ControlFlow\CancelingExample.cs



# Canceling Parallel.For (2)

```
Parallel.Invoke(
    () =>
    {
        Thread.Sleep(10);
        Console.WriteLine("Cancelling operation via CancellationToken.Cancel...");
        cts.Cancel();
    },
    () =>
    {
        try
        {
            Thread.Sleep(1);
            int[] nums = Enumerable.Range(0, 1000000).ToArray();
            Parallel.ForEach(nums, po, (num) =>
            {
                double d = Math.Sqrt(num) * Math.Sqrt(num * num);
            });
            Console.WriteLine("Operation completed without being cancelled.");
        }
        catch (OperationCanceledException e)
        {
            Console.WriteLine(e.Message);
            Assert.IsTrue(token.IsCancellationRequested);
        }
    });
```

# Multiple Exit Strategies

- Unhandled exceptions take priority over **Stop**, **Break**, or cancellation requests
- If no exceptions occurred but the **CancellationToken** was signaled and either **Stop** or **Break** was used
  - there's a potential race as to whether the loop will notice the cancellation prior to exiting:
  - If it does, the loop will exit with an **OperationCanceledException**.
  - If it doesn't, it will exit due to the **Stop/Break**



# Multiple Exit Strategies

- **Stop** and **Break** may not be used together. If they are, an exception will be raised.
- For long running iterations, there are multiple properties an iteration might want to check to see whether it should bail early:
  - **IsStopped**, **LowestBreakIteration**, **IsExceptional**
  - **ShouldExitCurrentIteration** property, which consolidates all of those checks in an efficient manner.

<http://code.msdn.microsoft.com/ParExtSamples>

- ParallelExtensionsExtras.csproj
  - Extensions/
    - AggregateExceptionExtensions.cs
    - CancellationTokenExtensions.cs
  - Partitioners/