Parallel and Asynchronous Programming with .NET

Dr. Dietrich Birngruber

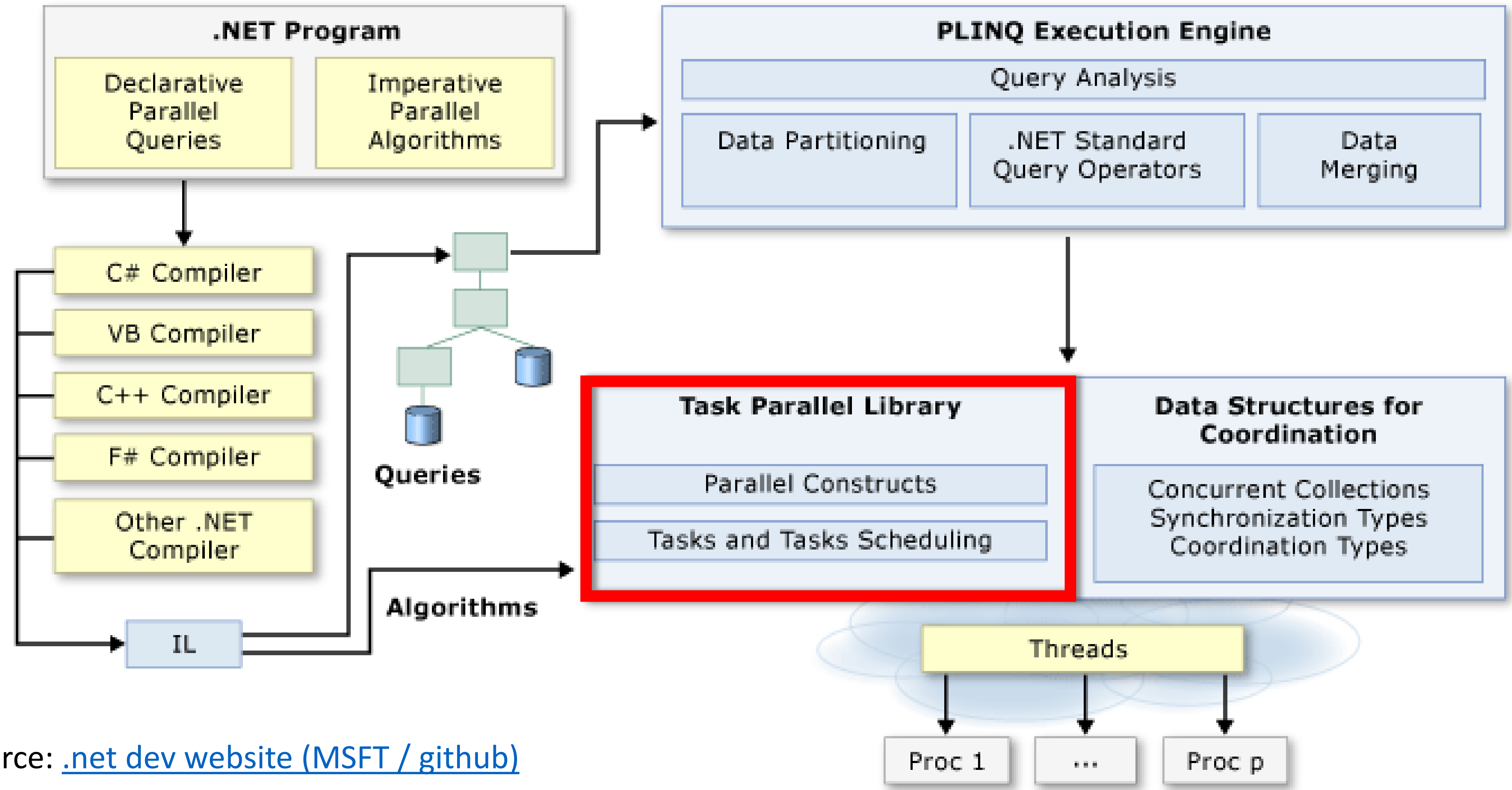
Parallel Programming Patterns in .NET

- Introduction
- Task Parallel Library (TPL)
- Parallel LINQ (PLINQ)
- Data Structures for Coordination
- Introduction to TPL Dataflow

Some Concepts

- Decompose
  - Your work into discrete units of work aka tasks
- Coordinate
  - These tasks as they run in parallel (e.g. join, fork)
- Share
  - The data needed to perform the tasks
- Potential Parallelism
  - Your program is written so that it runs faster when parallel hardware is available → parallel execution is not guaranteed!

Parallel Programming in .NET - Overview



Source: [.net dev website (MSFT / github)]

Parallelize My FOR Loop – Basic Idea

- Assumption / Precondition: many core CPU
- Decomposition: one iteration step is a discrete unit of work
  - body of the for-loop is split evenly between all cores
  - we could use one thread per core
- Coordination: Wait until all iteration steps are executed
- Share: the current step (counter-variable of type int)

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

Demo

My Parallel FOR Loop
(AsyncConsoleApp Demo02)

MyParallelFor: Pro / Cons

- Pro: dedicated threads exist purely to process our code

- .NET 4.x: Costs of a thread: 1 MB RAM → OutOfMemoryException
- .NET core : extra layer & configuration → prevent memory exception

- Costs of a thread: start / stop time
- Oversubscription: reuse our loop in another async method
  - T1 creates T2 creates T3 …
  - Thread context switches are expensive

MyParallelFor: Improvements?

- Compensation: we could use **ThreadPool** class
  - Faster start-up / managed by OS (or .net core)
  - if workload is not equivalent for each iteration → how to balance load?
  - Limits for available threads
- Balancing by partitioning in dynamic iterations / thread
  - E.g.: threads compete for iterations, instead of static iterations splitting
  - Requires more load balancing work & code → more code executed!
- Spectrum of partitioning tradeoffs
  - Fully static partitioning — Less Synchronization — vs. — Fully dynamic partitioning — More Load Balancing

Task Parallel Library (TPL)

- Class **Parallel**, since .NET 4.0, supports delightfully parallel loops
- Parallel.**For**, Parallel.**ForEach**<TSource>
- Some highlights:
  - Use ThreadPool under the hoods
  - Exception handling
  - Efficient load balancing
  - Handles nested parallelism (Parallel.For -> Parallel.For)
  - Early breakout of a loop

Demo

Parallel.For

(Parallel Demo02ParallelFor)

Exception handling: AggregateException

```
try
{
  Parallel.For(1, 10, i =>
                          {
                                  Console.WriteLine("Step {0}", i);
                                  if (i > 5) { throw new Exception("bäh"); }
                          });
}
catch (AggregateException ae)
{
  Console.WriteLine("InnerExceptions Count: {0}", ae.InnerExceptions.Count());
}
```

Fork / Join Pattern

- **Parallel.Invoke** executes given delegates in parallel (*FORK*)
- Waits for completion of the given delegates (*JOIN*)
- class Parallel takes care of propagating exceptions and task scheduling

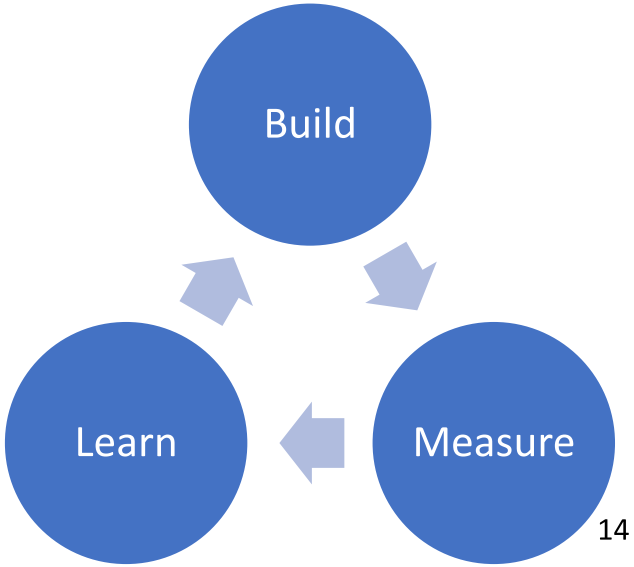
```
Parallel.Invoke (
        () => ComputeMean(),
        () => ComputeMedian(),
        () => ComputeMode() );
```

Parallel Anti-Patterns & Potential Pitfalls

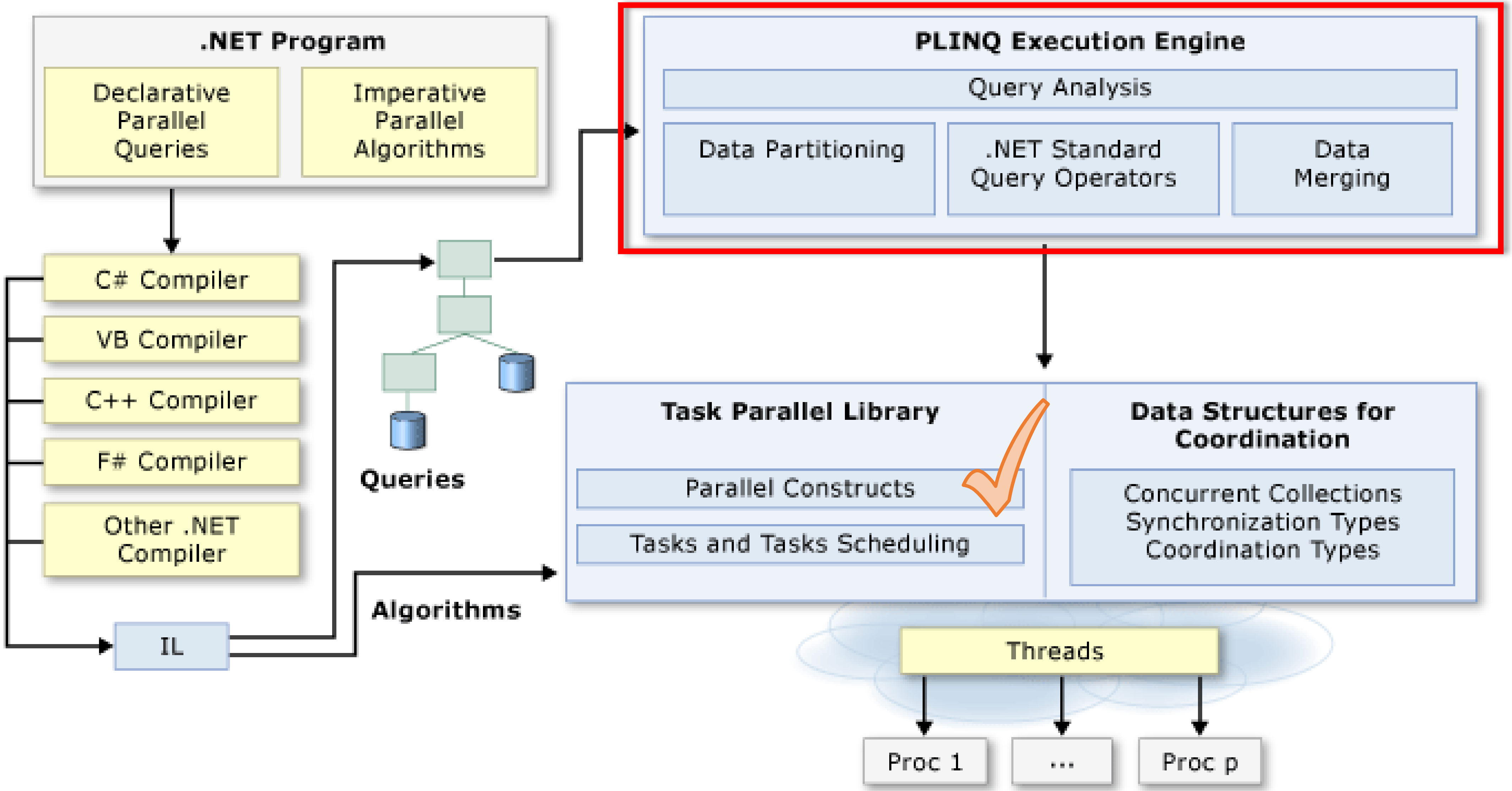
- Avoid shared data
  - don’t share data between iteration steps
  - Use System.Threading.ThreadLocal<T> to share state between steps
  - If you update a globally shared data: code for concurrent access is needed
- No downward iteration
  - often dependencies between iteration steps
  - `for(int i=upperBound-1; i>=0; --i) { /*...*/ }`
- No stepped iterations in for-loop
  - Not directly supported in the overloaded methods

Parallel Anti-Patterns & Potential Pitfall

- Avoid very ‘small / short’ loop bodies
  - There is overhead for creating and calling parallel code
  - But: too coarse-grained steps is also bad (no partitioning possible)
- Avoid executing Parallel loops on the UI thread
  - Caller waits for response → UI is not responsive anymore!
- Don’t assume Parallel.For is always faster
  - Build – measure performance – learn (= optimize)



Parallel Programming in .NET Overview



PLINQ

- Parallel implementation of LINQ to Objects
- Classes **ParallelEnumerable** and **ParallelQuery<T>**
  - Transform a IEnumerable<T> by calling extension method AsParallel() to a ParallelQuery<T>

```
IEnumerable<int> source = Enumerable.Range(1, 10000);

// Opt-in to PLINQ with AsParallel()
var evenNums = from num in source.AsParallel()
               where Compute(num) > 0
               select num;
```

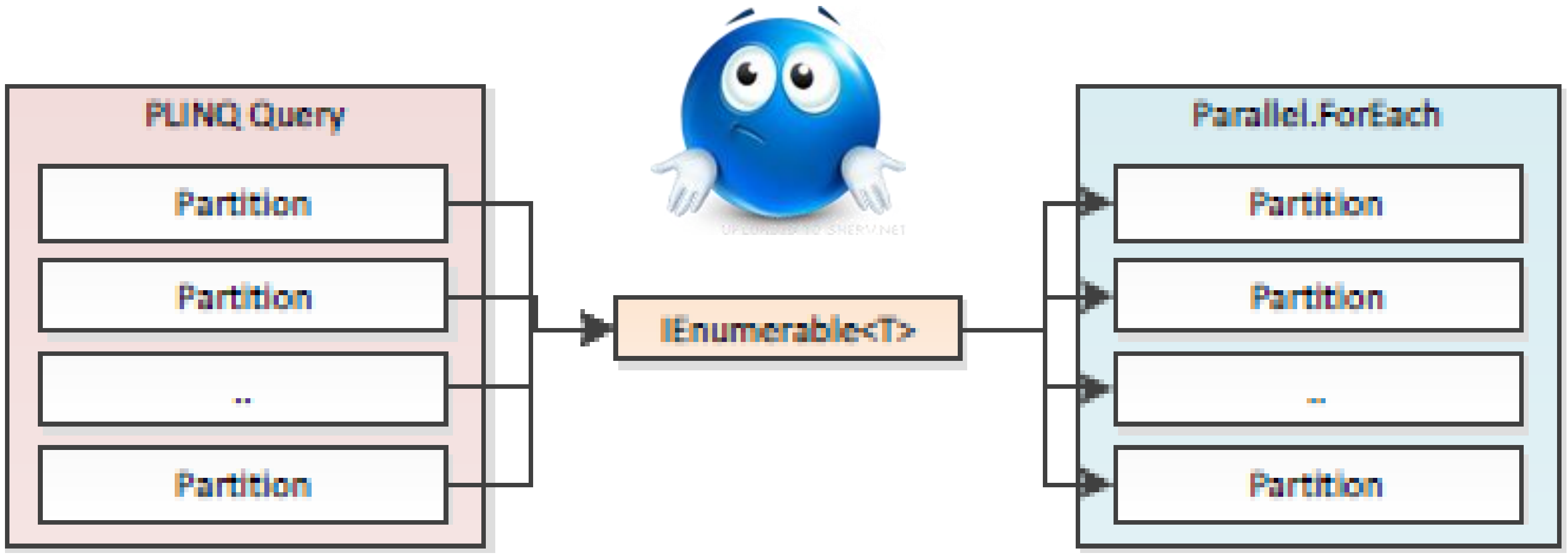
Demo

PLINQ

(Parallel Demo03)

PLINQ and Parallel – Don't mix!

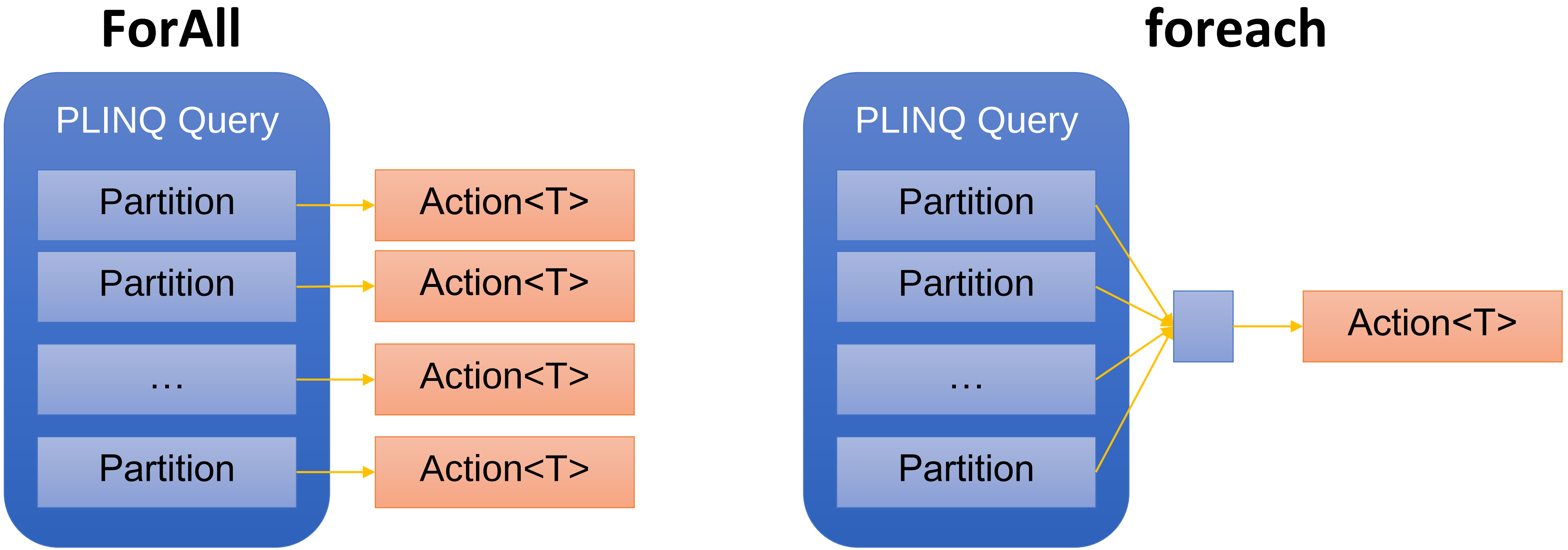
- **Don't mix** Parallel loop and ParallelQuery
  - PLINQ partitions → streams to IEnumerable → Parallel partitions again



- Use **ParallelEnumerable.ForAll** to iterate a ParallelQuery<T>

ForAll Method vs foreach Statement

- Iterating a ParallelQuery with **ParallelEnumerable.ForAll** or **foreach**



PLINQ Anti-Patterns and Pitfalls

- Small computational cost / short running methods
  - The more computational expensive, the greater the opportunity for speedup

```
var queryA = from num in numberList.AsParallel()
             select ExpensiveFunction(num); //good for PLINQ

var queryB = from num in numberList.AsParallel()
             where num % 2 > 0
             select num; //not good for PLINQ
```

PLINQ Anti-Patterns and Pitfalls

- Avoid over-parallelization

```
var q = from cust in customers.AsParallel()
        from order in cust.Orders.AsParallel()
        where order.OrderDate > date
        select new { cust, order };
```

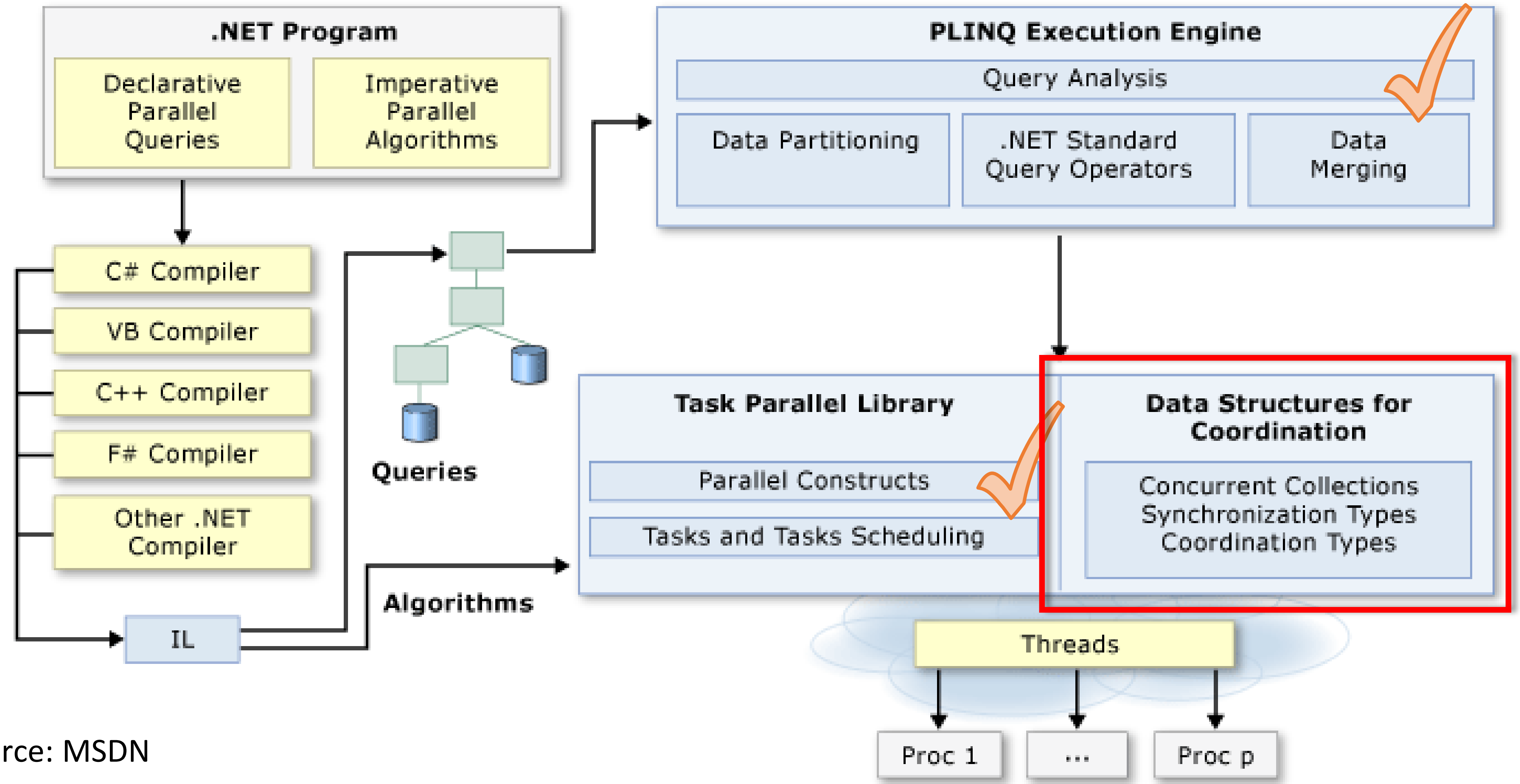
- Avoid unnecessary ordering operations
  - Parallel ordering is possible → ParallelEnumerable.AsOrdered

```
var orderedCities =    from city in cities.AsParallel().AsOrdered()
                       where city.Population > 10000
                       select city;
```

PLINQ Anti-Patterns and Pitfalls

- Avoid calls to non-thread-safe methods
- Limit calls to thread-safe methods
  - lock(syncObj) is expensive
  - Use class ReadWriteLockSlim
- Avoid accessing shared memory
  - E.g. static variables

Parallel Programming in .NET Overview



Source: MSDN

Data Structures for Coordination

Collection classes aka Thread-safe collections

- System.Collections.Concurrent.**BlockingCollection**<T>
- Concurrent**Bag**<T>
- Concurrent**Dictionary**<TKey, TValue>
- Concurrent**Queue**<T>
- Concurrent**Stack**<T>

Demo

Producer / Consumer with BlockingCollection

Data Structures for Coordination

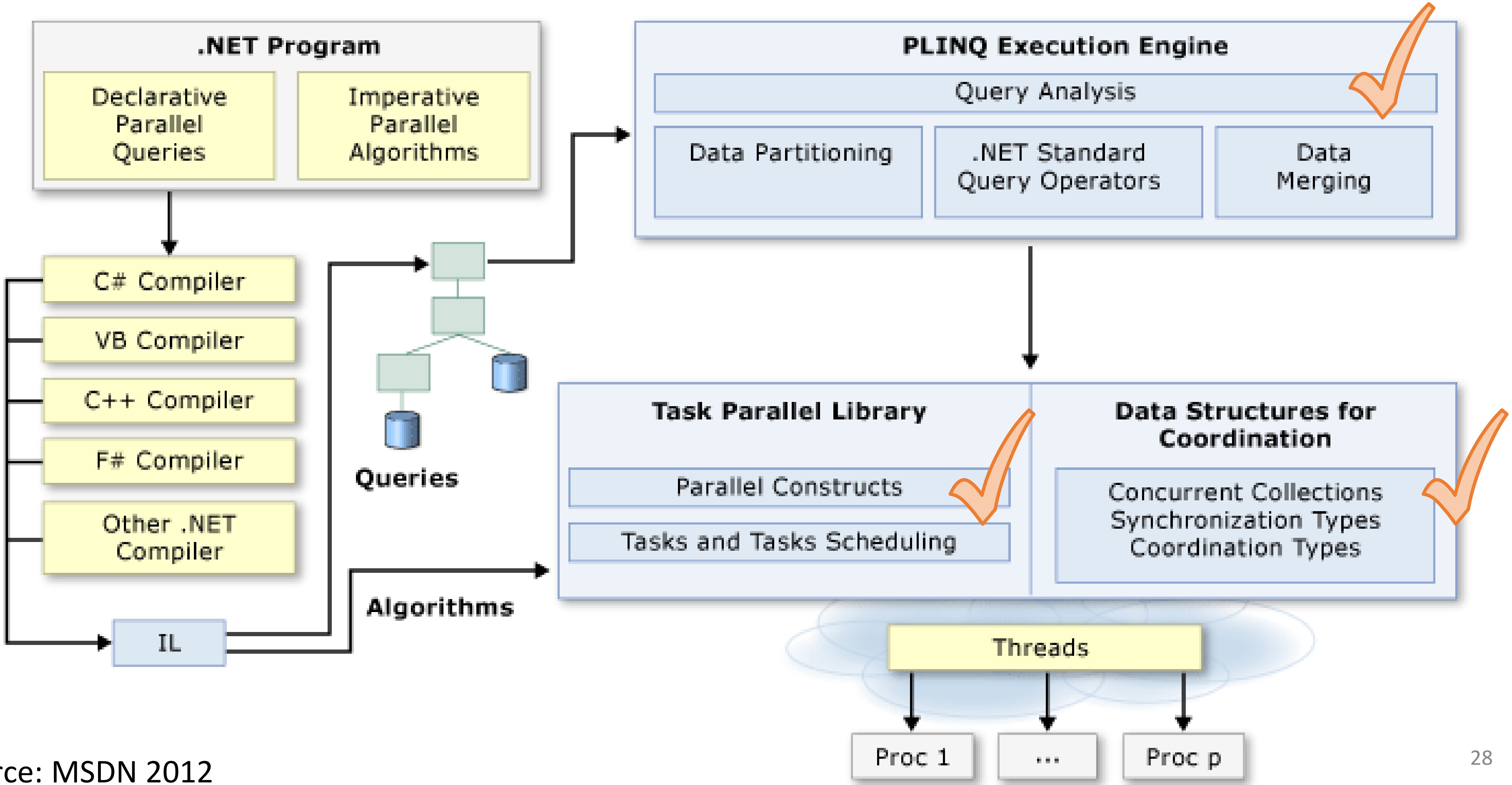
- Same data structures used for coordinating Threads, such as
  - System.Threading.**Barrier** class
    - Enables multiple threads to work on an algorithm in parallel by providing a point at which each task can signal its arrival and then block until some or all tasks have arrived.
  - **CountdownEvent**
    - Simplifies fork and join scenarios by providing an easy rendezvous mechanism.
  - **ManualResetEventSlim**
  - **SemaphoreSlim**
    - A synchronization primitive that limits the number of threads that can concurrently access a resource or a pool of resources.

Data Structures for Coordination

Lazy initialization classes:

- System.**Lazy**<T>
  Provides lightweight, thread-safe lazy-initialization – i.e. deferring the creation of large / resource intensive objects.
- System.Threading.**ThreadLocal**<T>
  Provides a lazily-initialized value on a per-thread basis, with each thread lazily-invoking the initialization function.
- System.Threading.**LazyInitializer**
  Provides static methods that avoid the need to allocate a dedicated, lazy initialization instance. Instead, they use references to ensure targets have been initialized as they are accessed.

Parallel Programming in .NET Overview



Source: MSDN 2012

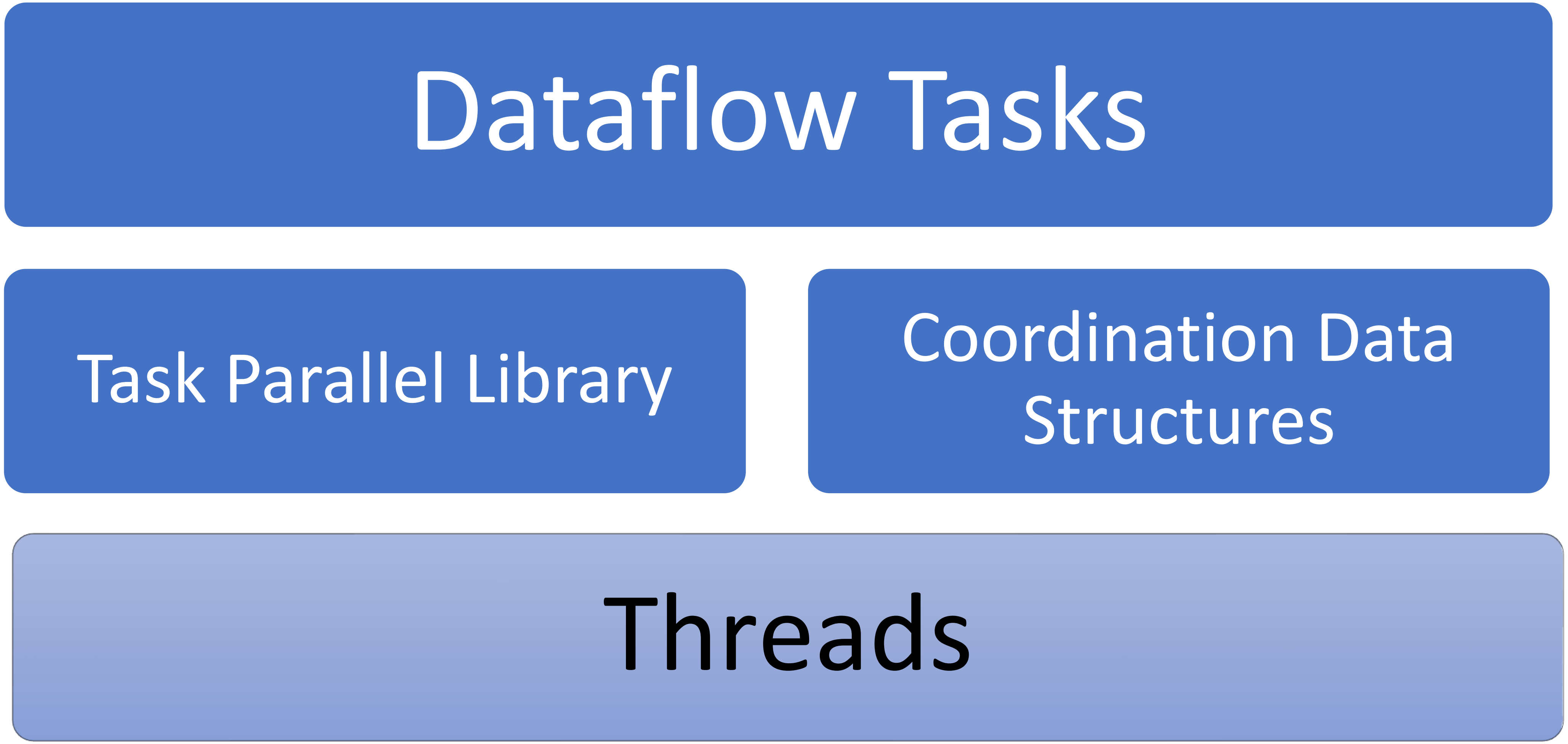
One more thing …

- Introduction to TPL Dataflow
- Hint: Take a look at Reactive Extensions (Rx)

TPL Dataflow (TDF)

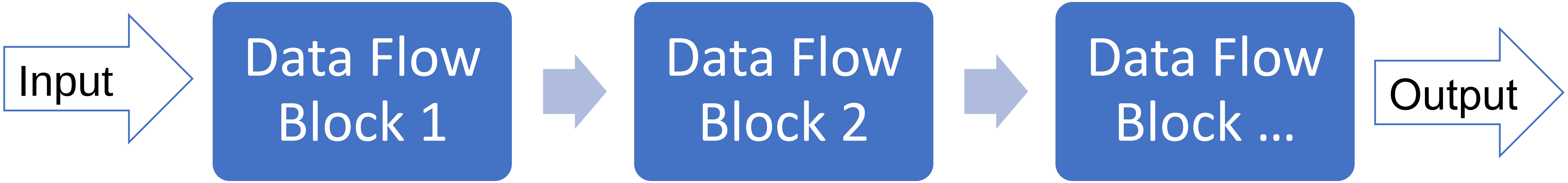
- TPL Dataflow (TDF) is a.NET library for building concurrent applications.
- It promotes actor/agent-oriented designs through primitives for in-process message passing, dataflow, and pipelining.
- TDF builds upon the APIs and scheduling infrastructure provided by the Task Parallel Library (TPL, since .NET 4), and integrates with the language support for asynchrony provided by C#, Visual Basic, and F#.

TPL Dataflow Introduction



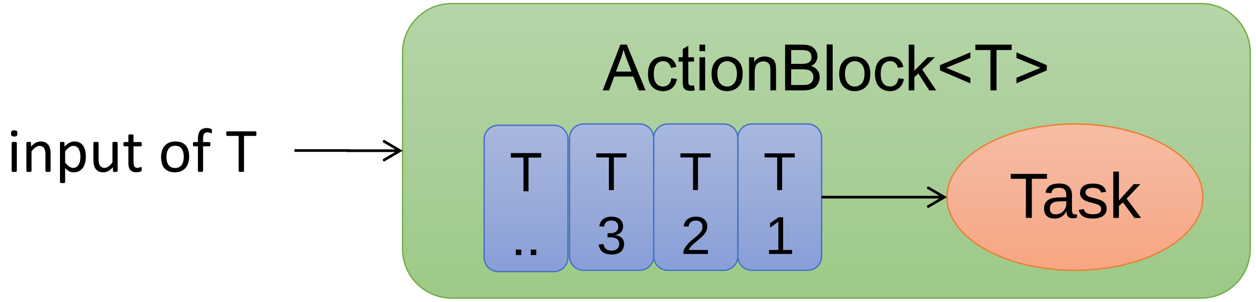
TDF Introduction

- Network of data flow blocks
- Like pipelines / consumer – producers / message processing agents



TDF (TPL Data Flow) Introduction

- A DataFlow block is a **message buffering** strategy with an async **Task** processing the messages
  - E.g. ActionBlock<T> which executes an action for each <T>
  - Degree of parallelism is configurable



Demo

Single ActionBlock

(Parallel Demo06)

TDF Introduction

- Data is always processed asynchronously
  - Tasks need not be scheduled explicitly
- TDF is a generator of tasks (like Parallel class and PLINQ)
  - Developers declaratively express data dependencies
- System.Threading.Tasks.Dataflow (NuGet package)

TDF Block Interfaces

- ISourceBlock<TOutput> generates messages
- ITargetBlock<TInput> processes messages
- IPropagatorBlock<in TInput, out TOutput> processes incoming messages and outputs the result to the next block
  - IPropagatorBlock inherts from ISourceBlock and ITargetBlock
  - Most of the predefined blocks are propagator blocks

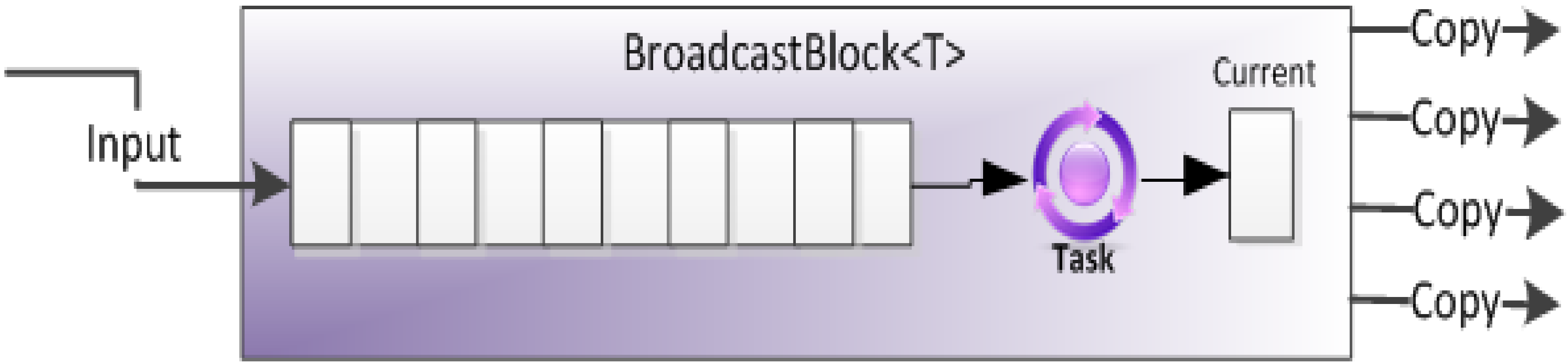
TDF Block Categories

There are three built –in TDF block categories:

- Pure buffering blocks
  - Buffer data in various ways
- Execution blocks
  - Execute user –provided code in response to data provided to the block
- Grouping blocks
  - Group data across one ore more source and under various constraints

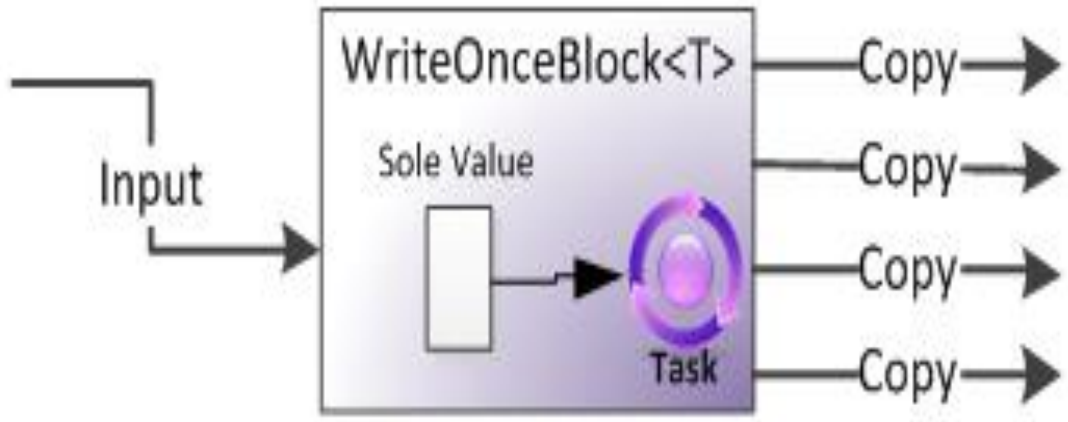
Pure Buffering Blocks

- BufferBlock<T>
  - Propagator for buffering instances of T for later consumption (FIFO queue)
  - Only one receiver
- BroadcastBlock<T>
  - All targets receive a copy by a user-provided cloning Fun<T,T>



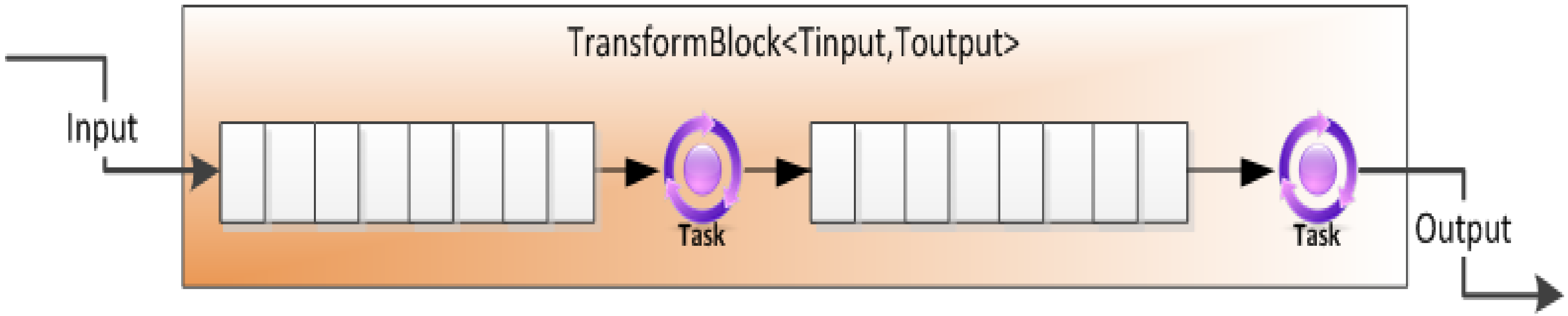
Pure Buffering Blocks

- WriteOnceBlock<T>
  - Similar to a read-only variable: write one value once in a time



Execution Blocks

- ActionBlock<T>
  - execution of a delegate to perform some action for each input
  - Can be used with synchronous and asynchronous action methods
- TransformBlock<TInput,TOutput>
  - Like ActionBlock but with an output Func<TInput, TOutput>

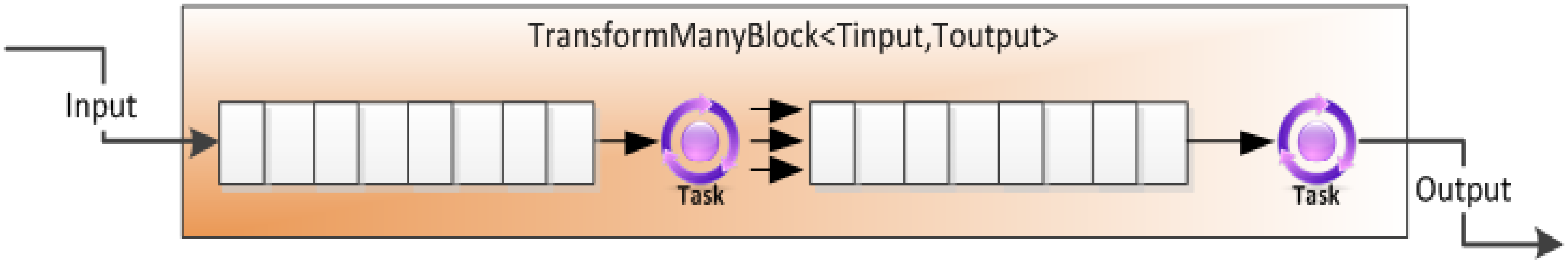


Execution Blocks

- TransformBlock code sample

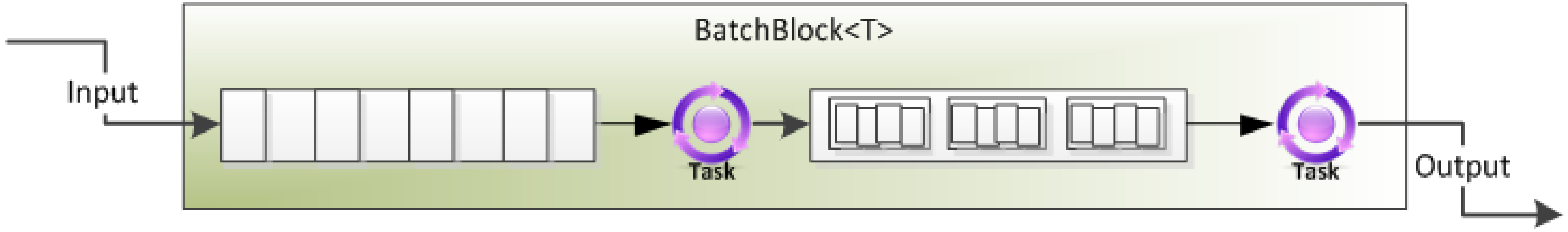
```
var compressor = new TransformBlock<byte[],byte[]>(input => Compress(input));
var encryptor = new TransformBlock<byte[],byte[]>(input => Encrypt(input));
compressor.LinkTo(encryptor);
```

- TransformManyBlock <TInput, TOutput>
  - Like TransformBlock but produces 1-N outputs for each input



Grouping Blocks

- BatchBlock<T>
  - combines N single items into one batch item, represented as an array of elements
  - Supports greedy and non greedy modes

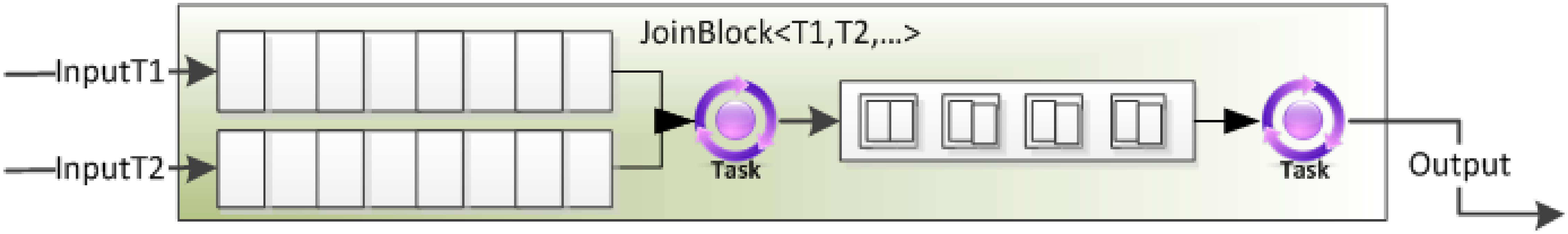


Grouping Blocks

- BatchBlock code exsample

```
var batchRequests = new BatchBlock<Request>(batchSize:100);
var sendToDb = new ActionBlock<Request[]>(reqs => SubmitToDatabase(reqs));
batchRequests.LinkTo(sendToDb);
```

- JoinBlock<T1,T2,…>
  - Group from multiple inputs to one output using Tuple<>



Hint: Reactive Extensions (Rx)

- Idea: use LINQ query operators with live data streams (e.g., events received from Azure Event Hub, or Kafka, - *not* Event Grid)
- Rx elevates live data streams to first-class citizens in .NET / JavaScript, allowing developers to react to data as it arrives
  - handle continuous stream of events
- It is built around the **IObservable<T>** (IAsyncObservable<T>) and **IObserver<T>** interfaces, enabling a push-based collection model
  - event is pushed into your code, when it becomes available
- To compose asynchronous and event-based programs declaratively
- TPL offers AsObserver / AsObservable extension methods
- GitHub - dotnet/reactive: The Reactive Extensions for .NET

Summary

- Task Parallel Library – Delightful parallel loops
- PLINQ – declarative, parallel linq queries
- Coordination data structures
- TDF – delightful parallel algorithms using data flow blocks, where data flows through a network of blocks