qDataflow

## Introduction

This document covers

Introduction

### What is it?

* Parallel stream processing library
* Supports dataflow, pipelining, and message-passing paradigms.
* Supports data buffering, batching and concurrent operations without locking.
* Implemented using Tasks, concurrent collections and other TPL core types.

### When to use it?

* System is organised around flows of data.
* Pipelining.
* High throughput, low latency demands.
* High computational demand
* Requirement to use multiple cores.

### What are the key features?

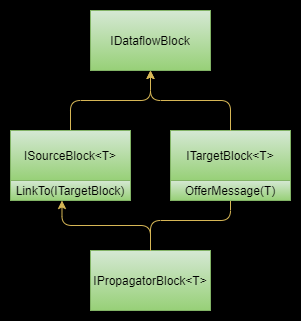
* Data dependencies are expressed declaratively.
* Data is always processed asynchronously.
* Developers do not need to explicitly schedule tasks.
* Computation expressed as dataflow graph or pipeline.
* Under the hood uses TPL tasks and concurrent collections.

### How does it differ from RX?

* RX composes and coordinates event streams using LINQ.
* TDF focuses on message passing and parallelization and buffering.

## Architecture

A TPL dataflow consists of blocks that buffer, process, and propagate data. Blocks can be connected into linear sequences known as pipelines or graphs known as networks. Blocks can be either sources, targets, or both. A block that is both a source and a target is known as a propagator. The library provides multiple build-int implementations that support different scenarios. While developers can develop their own blocks, the idea is the build it blocks will cover most common scenarios. The hierarchy looks as follows.



Sources can be linked to zero or more targets and targets can be linked from zero or more sources. This allows us to construct networks where sources automatically and asynchronously propagate data to targets. Sources are linked to target blocks using the LinkTo method. Sources pass data to targets by invoking the target’s OfferMessage method. The source and target can participate in a two-phase commit protocol

## Built-in blocks

The language provides a set of built-in blocks that are intended to cover most scenarios. These blocks handle asynchrony in a thread-safe manner so the consumer does not have to. **They enable multiple forms of data buffering, greedy and non-greedy receiving, joining, and batching**.

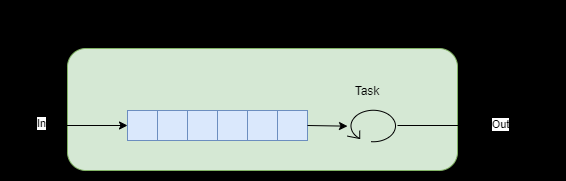
The built-in blocks fall into one of three categories.

* Pure Buffering Blocks
* Execution Blocks
* Grouping Blocks

### Buffering Blocks

#### BufferBlock<T>

The BufferBlock is a propagator that provides support for unbounded or bounded buffering of messages of type T. Posting to the block causes values to be stored in FIFO order by the block. The BufferBlock is one of the fundamental blocks when it comes to facilitating message passing.



##### Post stores values in FIFO order

Call Post on the block causes values to be stored in FIFO order.

var b = new BufferBlock<int>();

b.Post(1);

b.Post(2);

WriteLine(b.Receive());

WriteLine(b.Receive());

WriteLine("All received");

>> 1

>> 2

>> All received

##### Receive blocks on a buffer with no values.

Calling Receive on a buffer with no values is a blocking operation. Note in the below sample “All received” is never written to the console.

var b = new BufferBlock<int>();

b.Post(1);

b.Post(2);

WriteLine(b.Receive());

WriteLine(b.Receive());

WriteLine(b.Receive());

WriteLine("All received");

>> 1

>> 2

##### ReceiveAsync does not block.

var b = new BufferBlock<int>();

b.Post(1);

b.Post(2);

WriteLine(b.ReceiveAsync());

WriteLine(b.ReceiveAsync());

WriteLine(b.ReceiveAsync());

WriteLine("Called ReceiveAsync 3 times");

Sleep(1000);

b.Post(3);

Initially we see.

>> 1

>> 2

*>>awaiting...*

>> Called ReceiveAsync 3 times

And after 2 seconds

1

2

3

Called ReceiveAsync 3 times

##### Synchronous Producer/Consumer

BufferBlock<int> buffer = new BufferBlock<int>();

var consumerTask = Task.Factory.StartNew(() =>

{

while(true)

{

int item = buffer.Receive();

WriteLine($"Consumed item {item}");

}

});

var producerTask = Task.Factory.StartNew(() =>

{

for (int i = 0; i < 10; i++)

{

buffer.Post(i);

Sleep(1000);

}

});

Task.WaitAll(consumerTask, producerTask);

##### Asynchronous Producer/Consumer

BufferBlock<int> buffer = new BufferBlock<int>();

var consumerTask = Task.Factory.StartNew(async () =>

{

while(true)

{

**int item = await buffer.ReceiveAsync();**

WriteLine($"Consumed item {item}");

}

});

var producerTask = Task.Factory.StartNew(() =>

{

for (int i = 0; i < 10; i++)

{

buffer.Post(i);

Sleep(1000);

}

});

Task.WaitAll(consumerTask, producerTask);

##### Asynchronous Producer/Consumer with Throttling

[STAThread]

public static void Main()

{

// Create a buffer with a bound

var producerConsumer = new ProducerConsumer(3);

Window w = new Window() { SizeToContent = SizeToContent.WidthAndHeight };

StackPanel s = new StackPanel();

Button b = new Button() { Content = "Consume" };

b.Click += async (object sender, RoutedEventArgs e)

=> await producerConsumer.Consume();

s.Children.Add(b);

Button c = new Button() { Content = "Produce" };

c.Click += async (object sender, RoutedEventArgs e)

=> await producerConsumer.Produce();

s.Children.Add(c);

w.Content = s;

w.Show();

Dispatcher.Run();

}

public class ProducerConsumer

{

private BufferBlock<int> \_buffer;

private int \_pcCnt = 0;

public ProducerConsumer(int capacity)

{

var options = new DataflowBlockOptions { BoundedCapacity = 3 };

\_buffer = new BufferBlock<int>(options);

// Fill buffer to capacity

for (\_pcCnt = 0; \_pcCnt < capacity; \_pcCnt++)

{

\_buffer.Post(\_pcCnt);

}

}

public async Task Consume() => WriteLine(await \_buffer.ReceiveAsync());

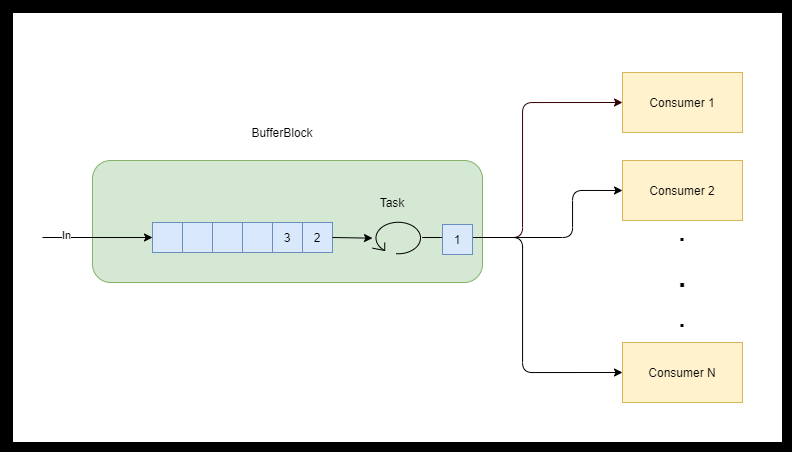
public async Task Produce() => WriteLine(await \_buffer.SendAsync(\_pcCnt++));

}

If the produce more than the capacity, no more will be accepted until the producer processes some data.

#### BroadcastBuffer<T>

The broadcast block broadcasts a copy of each value to all linked targets.



var b = new BroadcastBlock<int>(i=>i);

var a1 = new ActionBlock<int>(x =>

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target1 Received {x}"));

var a2 = new ActionBlock<int>(x =>

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target2 Received {x}"));

b.LinkTo(a1);

b.LinkTo(a2);

b.Post(1);

b.Post(2);

>> Thread 12, Target2 Received 1  
>> Thread 12, Target2 Received 2  
>> Thread 5, Target1 Received 1  
>> Thread 5, Target1 Received 2

The broadcast block always holds the last item it was offered. When any subsequent targets are added they are instantly given than item.

var b = new BroadcastBlock<int>(i=>i);

var a1 = new ActionBlock<int>(x =>

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target1 Received {x}"));

b.LinkTo(a1);

b.Post(1);

b.Post(2);

Sleep(100);

var a2 = new ActionBlock<int>(x =>

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target2 Received {x}"));

b.LinkTo(a2);

### WriteOnceBlocl<T>

Stores at most one value. Once a value has been it is immutable. Has broadcast semantics.

var b = new WriteOnceBlock<int>(i=>i);

var a1 = new ActionBlock<int>(x =>

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target1 Received {x}"));

b.LinkTo(a1);

b.Post(1);

b.Post(2);

Sleep(100);

var a2 = new ActionBlock<int>(x =>

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target2 Received {x}"));

b.LinkTo(a2);

>> Thread 10, Target1 Received 1  
>> Thread 9, Target2 Received 1

### Execution Blocks

#### ActionBlock<T>

Consider the simple ActionBlock<T>. This simple class is a target block. It provides an abstraction that encapsulates a buffer and tasks that process data from that buffer. The consumer uses a delegate to specify the logic to apply to each data item.

Consider the following fragment.

var ab = new ActionBlock<int>(x=> {

Thread.Sleep(1000);

Console.WriteLine($"Thread {Thread.CurrentThread.ManagedThreadId} Got {x}");

});

for (int i = 0; i < 3; i++)

{

Console.WriteLine($"Thread {Thread.CurrentThread.ManagedThreadId} Sending {i}");

ab.Post(i);

}

>> Thread 1 Sending 0  
>> Thread 1 Sending 1  
>> Thread 1 Sending 2  
>> Thread 14 Got 0  
>> Thread 14 Got 1  
>> Thread 14 Got 2

Notice how the action block action is being processed on a background thread because internally the action block uses tasks. Rather than an Action<T> delegate we can provide a Func<TInput, Task> so we can provide async functions.

Func<int, Task> f = (x) =>

{

return Task.Factory.StartNew(() =>

{

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target1 Received {x}");

});

};

var a = new ActionBlock<int>(f);

a.Post(1);

>> Thread 9, Target1 Received 1

Action blocks have buffering built in.

Func<int, Task> f = (x) =>

{

return Task.Factory.StartNew(() =>

{

Sleep(1000);

WriteLine($"Thread {CurrentThread.ManagedThreadId}, Target1 Received {x}");

});

};

var a = new ActionBlock<int>(f);

a.Post(1);

a.Post(2);

a.Post(3);

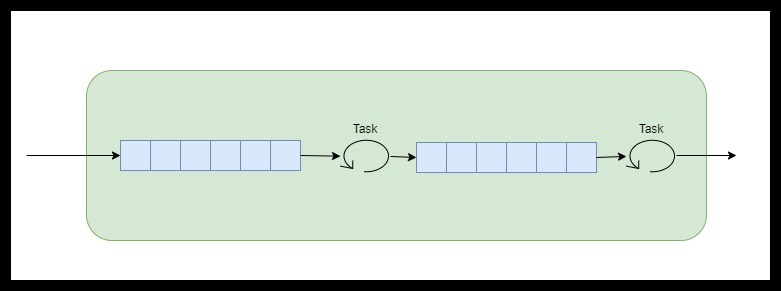
WriteLine($"Thread {CurrentThread.ManagedThreadId},Published three values");

>> Thread 1,Published three values  
>> Thread 13, Target1 Received 1   
>> Thread 13, Target1 Received 2  
>> Thread 13, Target1 Received 3

### Transform Blocks

#### TransformBlock<T>

Transform blocks transforms data so it is both a source and a target. The transform buffers inputs and outputs. The output buffer behaves like a BufferBlock.



If the transform is configured to process messages asynchronously the transformed values are still delivered to the output block in the order the input values arrived (internally uses a reordering buffer).

#### TransformManyBlock<T>

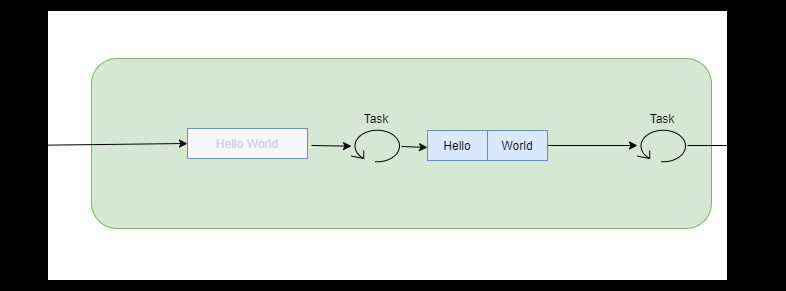
Func<string,string[]> split = (input) => input.Split(" ");

var tm = new TransformManyBlock<string,string>(split);

tm.Post("Hello World");

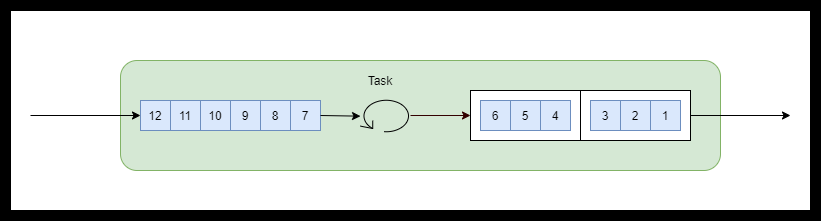
WriteLine($"{tm.Receive()}");

WriteLine($"{tm.Receive()}");



### Combining Blocks

#### BatchBlock<T>



var a = new BatchBlock<int>(3);

for (int i = 0; i < 6; i++)

{

a.Post(i);

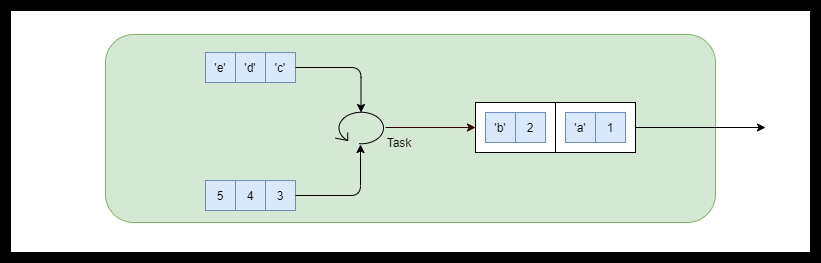
}

a.Receive().Dump();

a.Receive().Dump();

|  |
| --- |
| **5**Int32[3] **44** |
| 0 |
| 1 |
| 2 |
| **5**Int32[3] **44** |
| 3 |
| 4 |
| 5 |

#### Join Block<T>



### Configuring Built-in blocks

### BufferBlock<T>

The buffer block is an unbounded buffer that supports asynchronous producer/consumer scenarios.

#### Post and Buffer Size

What happens if we post to an action whose buffer is full? If we post the messages will be discarded.

var ab = new ActionBlock<int>(x=> {

Thread.Sleep(1000);

WriteLine($"Thread {Thread.CurrentThread.ManagedThreadId} Got {x}");

}, new ExecutionDataflowBlockOptions() {BoundedCapacity=3});

for (int i = 0; i < 5; i++)

{

WriteLine($"Thread {Thread.CurrentThread.ManagedThreadId} Sending {i} {(ab.Post(i)

? "accepted"

: "rejected")}");

}

>> Thread 1 Sending 0 accepted  
>> Thread 1 Sending 1 accepted  
>> Thread 1 Sending 2 accepted  
>> Thread 1 Sending 3 rejected  
>> Thread 1 Sending 4 rejected  
>> Thread 22 Got 0  
>> Thread 22 Got 1  
>> Thread 22 Got 2

#### SendAsync and Buffer Size

SendAsync gives us more complex behaviour. The documentation mentions ‘postponement’ It seem if the target offers postponement, then the send async method will buffer until the call for until the target is ready.

var ab = new ActionBlock<int>(x=> {

Sleep(1000);

WriteLine($"Thread {CurrentThread.ManagedThreadId} Got {x}");

}, new ExecutionDataflowBlockOptions() {BoundedCapacity=2});

for (int i = 0; i < 4; i++)

{

WriteLine($"Thread {CurrentThread.ManagedThreadId} Sending {i} {(await ab.SendAsync(i) ? "accepted" : "rejected")}");

}

In pipelines and networks, sources asynchronously propagate data to targets as and when it becomes available. A source can link to zero or more targets and a target can be linked from zero or more sources. This is performed using the ISourceBlock<T>.LinkTo method. If we use the predefined dataflow blocks, we can safely add and remove blocks from a pipeline or network concurrently as they handle threading safely. The following shows a simple pipeline.

var square = new TransformBlock<int,int>(n => n \* n);

var doubleIt = new TransformBlock<int,int>(n => 2 \* n);

var increment = new TransformBlock<int,int>(n => n +1 );

var print = new ActionBlock<int>(n => Console.WriteLine(n));

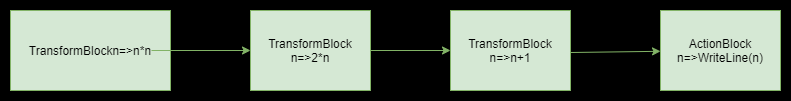
square.LinkTo(doubleIt);

doubleIt.LinkTo(increment);

increment.LinkTo(print);

square.Post(2);

>> 9



## Predefined Blocks

The language comes with a set of predefined blocks which are divided into three categories.

* Buffering Blocks
* Execution Blocks
* Grouping Blocks

### Buffering Blocks

“The TPL Dataflow Library provides a foundation for message passing and parallelizing CPU-intensive and I/O-intensive applications that have high throughput and low latency. It also gives you explicit control over how data is buffered and moves around the system”

* Microsoft Documentation

Support coarse-grained dataflow and pipelining. The pipelines consist of dataflow blocks that buffer and process data.

The library comes with predefined implementations of these interfaces.

nnection multiplexing. Concurrent requests across a single TCP connection.