The Future of Parallel Programming in the .NET Framework

Danny Shih
Microsoft Corporation
Program Manager
DISCLAIMER

▪ This is a talk about the future…
  – All content is subject to change.
  – The technology being discussed…
    ▪ …is almost entirely available in CTP form NOW.
    ▪ … but may never actually ship (we’re doing the best we can).
AGENDA

- Present
  - Recap of parallel programming in the .NET Framework 4
- Future
  - Visual Studio Async
  - TPL Dataflow
RECAP PARALLEL PROGRAMMING IN .NET 4

- Feature Areas
  - Task Parallel Library (TPL)
  - Parallel LINQ (PLINQ)
  - Thread-safe data structures and synchronization primitives
- Pure .NET libraries

```csharp
ConcurrentQueue<int> cq = new ConcurrentQueue<int>();

// Thread 1    // Thread 2
cq.Enqueue(42);  cq.Enqueue(7);

var results = from i in cq
               where Is
              orderby  
               select C
```
AGENDA CHECKPOINT

- Present
  - Recap of parallel programming in the .NET Framework 4
- Future
  - Visual Studio Async
  - TPL Dataflow
**VISUAL STUDIO ASYNC**

- **Application Trends**
  - Increasingly connected
    - More latency (e.g. everything as a service)
    - More UI responsiveness problems
    - More scalability issues
  - User -> =( 
// Synchronous
TResult Foo(...);

// Asynchronous Programming Model (APM)
IAsyncResult BeginFoo(..., AsyncCallback callback, object state);
TResult EndFoo(IAsyncResult asyncResult);

// Event-based Asynchronous Pattern (EAP)
public void FooAsync(...);
public event EventHandler<FooCompletedEventArgs> FooCompleted;
public void CopyStreamToStream(Stream source, Stream destination) {
    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = source.Read(buffer, 0, buffer.Length)) != 0) {
        destination.Write(buffer, 0, numRead);
    }
}
public void CopyStreamToStream(Stream source, Stream destination)
{
    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = source.Read(buffer, 0, buffer.Length)) != 0)
    {
        destination.Write(buffer, 0, numRead);
    }
}

public IAsyncResult BeginCopyStreamToStream(Stream source, Stream destination)
{
    var tcs = new TaskCompletionSource<object>();
    byte[] buffer = new byte[0x1000];
    Action<IAsyncResult> readWriteLoop = null;
    readWriteLoop = iar =>
    {
        try
        {
            for (bool isRead = iar == null; isRead = !isRead)
            {
                switch (isRead)
                {
                    case true:
                        iar = source.BeginRead(buffer, 0, buffer.Length, readResult =>
                        { if (!iar.CompletedSynchronously)
                            return;
                        }, null);
                        if (!iar.CompletedSynchronously)
                            return;
                        break;
                    case false:
                        int numRead = source.EndRead iar;
                        if (numRead == 0)
                        {
                            tcs.TrySetResult(null);
                            return;
                        }
                        iar = destination.BeginWrite(buffer, 0, numRead, writeResult =>
                        { if (!iar.CompletedSynchronously)
                            return;
                        }, null);
                        destination.EndWrite iar;
                        break;
                }
            }
            catch (Exception e) { tcs.TrySetException(e); }
        };
    }
    return tcs.Task;
}

public void EndCopyStreamToStream(IAsyncResult asyncResult)
{
    ((Task)asyncResult).Wait();
}
public void CopyStreamToStream(Stream source, Stream destination)
{
    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = source.Read(buffer, 0, buffer.Length)) != 0)
    {
        destination.Write(buffer, 0, numRead);
    }
}

public async Task CopyStreamToStreamAsync(Stream source, Stream destination)
{
    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await source.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        await destination.WriteAsync(buffer, 0, numRead);
    }
}
VISUAL STUDIO ASYNC – LANGUAGE AND FRAMEWORK

Language
- “async” modifier marks a method or lambda
- “await” operator yields control until the awaited Task completes

Framework
- Task and Task<TResult> represent ongoing operations
  - E.g. Async I/O, background work, etc.
  - Single object for status, result, and exceptions
- New APIs round out the experience
// Synchronous
TResult Foo(...) ;

// Asynchronous Programming Model (APM)
IAAsyncResult BeginFoo(..., AsyncCallback callback, object state);
TResult EndFoo(IAAsyncResult asyncResult);

// Event-based Asynchronous Pattern (EAP)
public void FooAsync(...);
public event EventHandler<FooCompletedEventArgs> FooCompleted;

// Task-based Asynchronous Pattern (TAP)
Task<TResult> FooAsync(...);
VISUAL STUDIO ASYNC

▪ DEMOS

▪ Visual Studio Async CTP:
AGENDA CHECKPOINT

- Present
  - Recap of parallel programming in the .NET Framework 4
- Future
  - Visual Studio Async
  - TPL Dataflow
TPL DATAFLOW – COMPLEMENTING PARALLEL PROGRAMMING IN .NET 4

- .NET parallel programming was proactive in nature
  - “Here’s the data. Now set up the computation.”
  - Primitives for task and data parallelism

- Missing the reactive piece
  - “Set up the computation. Now here’s the data.”
  - Primitives for dataflow parallelism
TPL DATAFLOW OVERVIEW

- Primitives for in-progress message passing
  - Blocks that can buffer and process data
  - Can be linked together to create networks
  - Useful in agent/actor paradigm

- Inspired by
  - Decades of computer science research/history
  - Related Microsoft technologies
    - Asynchronous Agents Library in Visual C++ 2010
    - CCR from Microsoft Robotics
    - Axum incubation project
```csharp
var c = new ActionBlock<int>(i =>
{
    Process(i);
});

for(int i = 0; i < 5; i++)
{
    c.Post(i);
}
```
TPL DATAFLOW BLOCK HIERARCHY

IDataflowBlock

ISourceBlock<TOutput>
(a source of data)

ITargetBlock<TInput>
(a target for data)

IPropagatorBlock<> (a source and target)

Built-in blocks for **Buffering and Propagation**

Built-in blocks for **Executing**

Built-in blocks for **Joining**
### Blocks for Buffering and Propagation

- **BufferBlock<T>**
  - Delivers each input element to at most 1 target

- **WriteOnceBlock<T>**
  - Accepts and buffers only 1 element, ever
  - Delivers the 1 element to all targets

- **BroadcastBlock<T>**
  - Broadcasts each input element to all targets
**BLOCKS FOR EXECUTING**

- **ActionBlock<TInput>**
  - Executes an action for each input element

- **TransformBlock<TInput, TOutput>**
  - Transforms each input element to an output element

- **TransformManyBlock<TInput, TOutput>**
  - Transforms each input element to a collection of output elements
**BLOCKS FOR JOINING**

- **BatchBlock\(<T>\)**
  - Batches multiple input elements into batches (arrays)

- **JoinBlock\(<T_1, T_2>\)**
  - Joins pairs of input elements into tuples

- **BatchedJoinBlock\(<T_1, T_2>\)**
  - Joins pairs of collections of input elements into tuples
THANK YOU!

- **CTPs**
  - VS Async: [http://msdn.com/async](http://msdn.com/async)

- **Forums**

- My email: [dashih@microsoft.com](mailto:dashih@microsoft.com)
Disclaimer & Attribution
The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

The contents of this presentation were provided by individual(s) and/or company listed on the title page. The information and opinions presented in this presentation may not represent AMD’s positions, strategies or opinions. Unless explicitly stated, AMD is not responsible for the content herein and no endorsements are implied.
OPTIONS AND UTILITY FUNCTIONS

DataflowBlockOptions
- TaskScheduler
- MaxMessagesPerTask
- CancellationToken
- BoundedCapacity

ExecutionDataflowBlockOptions
- MaxDegreeOfParallelism

GroupingDataflowBlockOptions
- Greedy
- MaxNumberOfGroups

DataflowBlock extensions methods and utility functions
- SendAsync
- Receive, ReceiveAsync
- Choose
- ToObservable/ToObserver
- Etc.