The Future of Parallel Programming in the .NET Framework

Igor Ostrovsky
Software Engineer
Microsoft Corporation
DISCLAIMER

- This is a talk about the {near} future...
  - All content is subject to change.
  - The technology being discussed...
    - ...is mostly available in CTP form now.
    - ...may never actually ship (but we’re doing the best we can to make it).
Agenda

Future

- Visual Studio Async
- TPL Dataflow
Visual Studio Async
Trends

Increasingly connected applications
- More latency (e.g. everything as a service)
- More UI responsiveness problems
- 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 (readResult.CompletedSynchronously) return;
                            readWriteLoop(readResult);
                        }, 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 (writeResult.CompletedSynchronously) return;
                            destination.EndWrite(writeResult);
                            readWriteLoop(null);
                        }, null);
                        if (!iar.CompletedSynchronously) return;
                        destination.EndWrite(iar);
                        break;
                }
            }
        } catch (Exception e) { tcs.TrySetException(e); }
    };
    readWriteLoop(null);
    return tcs.Task;
}

public void EndCopyStreamToStream(IAsyncResult asyncResult) {
    ((Task<AsyncResult>).Wait());
}
Visual Studio Async
Your asynchronous code with the Visual Studio Async CTP...

```csharp
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
Tasks and Language

Language
- “async” modifier marks method or lambda as asynchronous
- “await” operator yields control until 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
Visual Studio Async

Related Additions

- Combinators
  - Task.WhenAll, Task.WhenAny
- Timer integration
  - Task.Delay(TimeSpan), CancellationTokenSource.CancelAfter(TimeSpan)
- Task scheduling
  - ConcurrentExclusiveSchedulerPair
- Fine-grained control
  - TaskCreationOptions.DenyChildAttach
  - EnumerablePartitionerOptions
- ThreadLocal.Values
// 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;

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

System.IO.Stream.ReadAsync(...);
  .WriteAsync(...);
  .FlushAsync();
  .CopyToAsync(...);
Visual Studio Async Demo

- DEMO – Sleeping on the UI
Agenda Checkpoint

- Future
  - Tasks and Language
  - TPL Dataflow
TPL Dataflow
Complementing Parallel Programming in .NET 4

- 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-process message passing
  - Blocks that can buffer and process data
  - Can be linked together to create networks

- 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
var c = new ActionBlock<int>(i =>
{
    Process(i);
});

for(int i = 0; i < 5; i++)
{
    c.Post(i);
}
TPL Dataflow
Blocks for Buffering and Propagation

- **BufferBlock\(<T>\)**
  - Buffers an unlimited number of elements
  - Delivers each element to at most 1 target

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

- **BroadcastBlock\(<T>\)**
  - Overwrites each element with the next (buffers until this happens)
  - Delivers each element to all linked targets
TPL Dataflow
Blocks for Executing

- ActionBlock<TInput>
  - Executes an Action<TInput> for each element
  - Buffers input until processed

- TransformBlock<TInput, TOutput>
  - Executes a Func<TInput, TOutput> for each element
  - Buffers input until processed and output until consumed

- TransformManyBlock<TInput, TOutput>
  - Executes a Func<TInput, IEnumerable<TOutput>> for each element
  - Buffers input until processed and output until consumed
TPL Dataflow
Blocks for Joining

- **BatchBlock**<sup>&lt;T&gt;</sup>
  - Groups multiple Ts into one T[]
  - Supports greedy and non-greedy

- **JoinBlock**<sup>&lt;T1, T2&gt;</sup>
  - Groups on T1 and one T2 to form a Tuple&lt;T1, T2&gt;
  - Supports greedy and non-greedy

- **BatchedJoinBlock**<sup>&lt;T1, T2&gt;</sup>
  - Groups T1s and T2s into one Tuple&lt;IList&lt;T1&gt;, IList&lt;T2&gt;&gt;
Related Content

- Parallel Programming Dev Center:

- Downloads

- Forums