LIVE! 360 TECH CON 2025
ORLANDO FLORIDA
TECH EVENTS WITH PERSPECTIVE

Patterns and Tools for

Parallel Programming in C#

Jeremy Clark
Developer Betterer
GitHub: jeremybytes

Level: Intermediate

Session Survey

- Your feedback is very important to us
- Please take a moment to complete the session survey found in the mobile app
- Use the QR code or search for “Converge360 Events” in your app store
- Find this session on the Agenda tab
- Click “Session Evaluation”
- Thank you!

LIVE! 360
TECH EVENTS WITH PERSPECTIVE

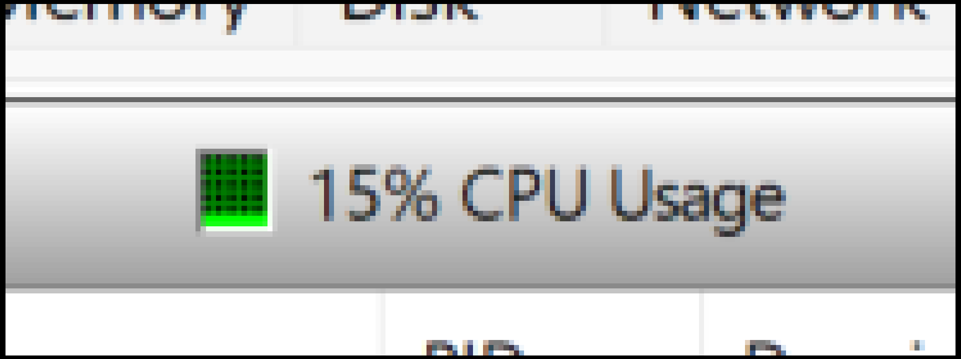
Patterns and Tools for

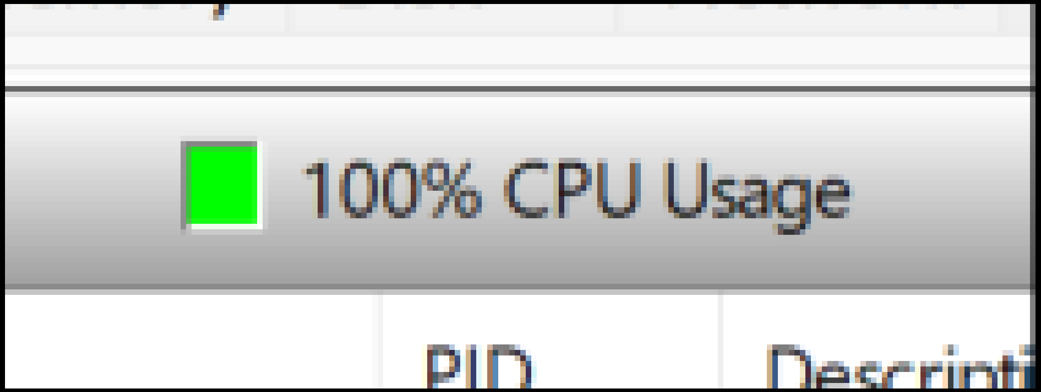
Parallel Programming in C#

Jeremy Clark
Developer Betterer
GitHub: jeremybytes

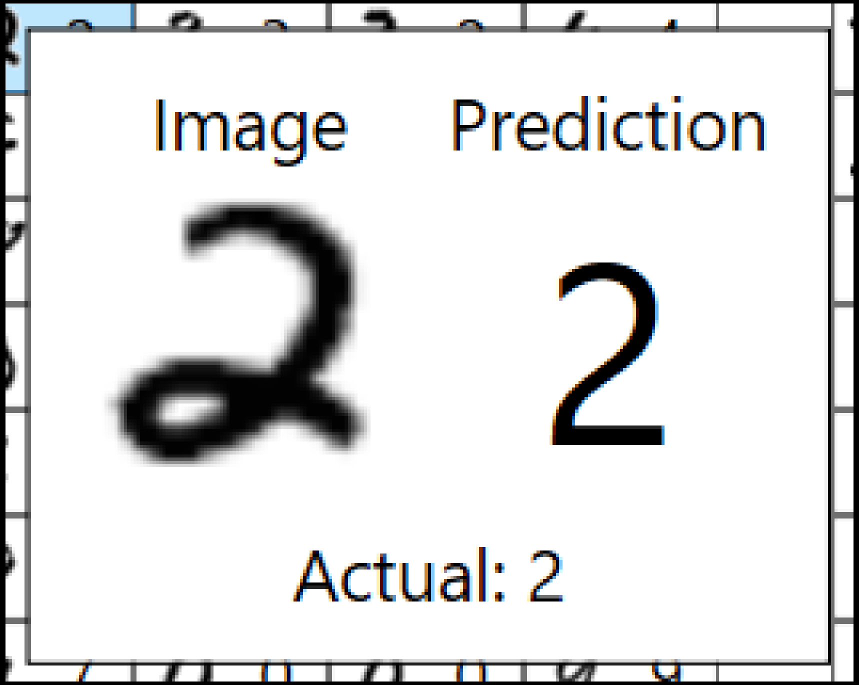
Level: Intermediate

Today's Goal

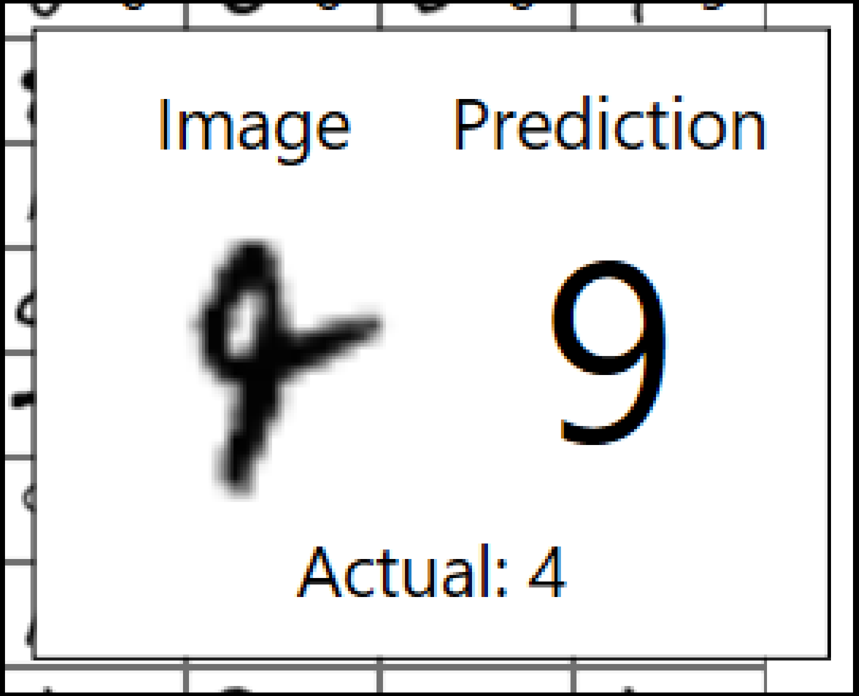




Recognizing Hand-written Digits



Correct



Incorrect

Parallel Tools & Considerations

Parallel Tools

- async / await
- Task with continuations
- Parallel.ForEachAsync
- Channel

Considerations

- Thread-safe Updates
- Parallel Continuations
- Continuing on the Main Thread
- Limiting Parallelism

Comparing Parallel Approaches

	Await	Task	ForEachAsync	Channel
Runs in Parallel	No	**Yes**	**Yes**	**Yes**
Continuation on Main Thread	**Yes**	**Yes** (optional)	No	**Yes**
Continuation in Parallel	No	**Yes**	**Yes**	No (optional)
Set Degrees of Parallelism	No	No	**Yes**	No

await is Sequential

- Multiple "await"'s run in sequence (one at a time)

```
await GetPerson(1);
await GetPerson(2);
await GetPerson(3);
```

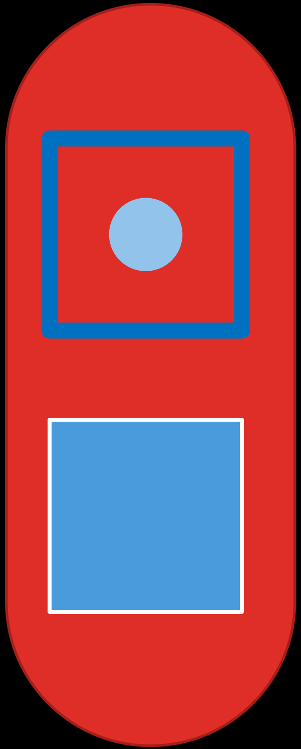
`GetPerson(2)` will not run until after `GetPerson(1)` is complete.

`GetPerson(3)` will not run until after `GetPerson(2)` is complete.

Looping await

- Each iteration at the loop with pause at the await
- The next iteration will not run until the previous is done

```
foreach (var imageData in validation)
{
    var prediction =
        await classifier.Predict(imageData);
    DisplayImages(prediction);
}
```



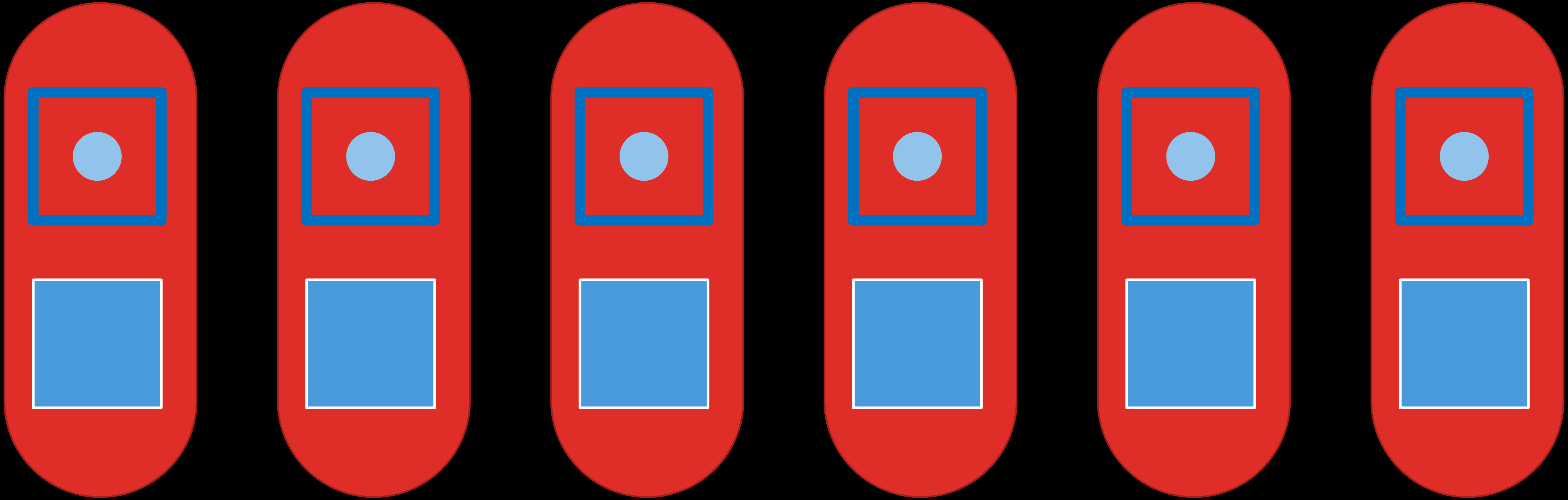
Parallel with Task

- Multiple non-awaited Tasks can run in parallel (at the same time)

```
GetPerson(1).ContinueWith(…);
GetPerson(2).ContinueWith(…);
GetPerson(3).ContinueWith(…);
```

`GetPerson(1)`, `GetPerson(2)`, and `GetPerson(3)` all run at the same time.

Get Data / Use Data



Parallel Loop with Task

- Task with a continuation does not pause the loop for each iteration
- The Tasks run in parallel

```
foreach (var imageData in validation)
{
    var predictionTask = classifier.Predict(imageData);
    predictionTask.ContinueWith(
        t => DisplayImages(t.Result));
}
```

Waiting for Parallel Tasks

- `await Task.WhenAll` can be used to determine when all tasks are complete.

```
List<Task> allTasks = new();
allTasks.Add(task1);
allTasks.Add(task2);
allTasks.Add(task3);
await Task.WhenAll(allTasks);
```

Parallel.ForEachAsync

- Loops over items and runs them in parallel
- "`await`" can be used safely inside the loop
  - Note: this is not true for "Parallel.ForEach"
- The entire loop can be "`await`"ed (this means all iterations will be complete)
- Available in .NET 8 / 9 / 10 (not .NET Framework)

Parallel.ForEachAsync

Waits for all iterations to finish

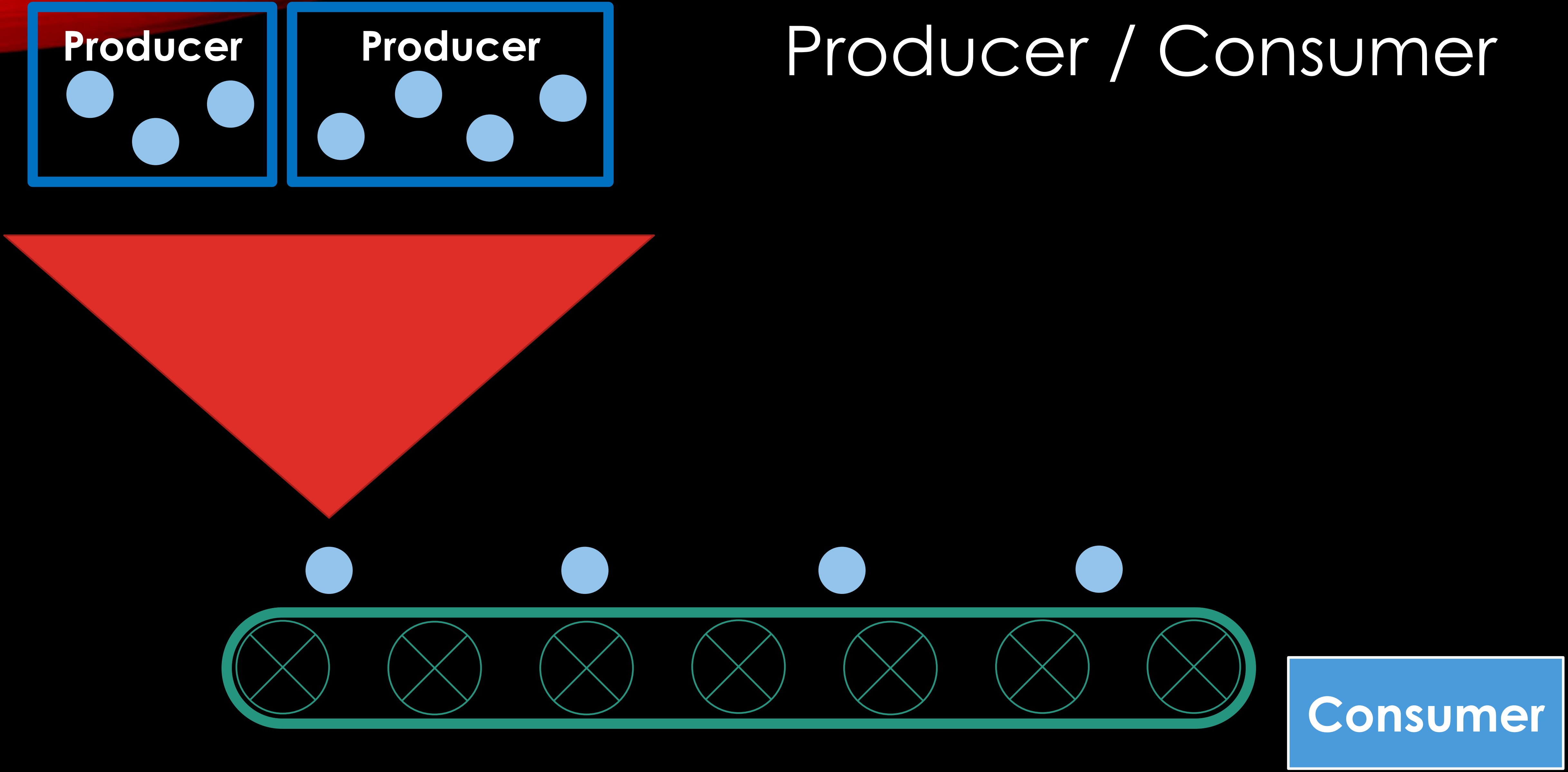
```
await Parallel.ForEachAsync(
    validation, ← The items to iterate over
    new ParallelOptions() { MaxDegreeOfParallelism = 10 },
    async (imageData, _) =>
    {
        var prediction =
            await classifier.Predict(imageData);
        DisplayImages(prediction);
    });
```

The method to run in parallel

ParallelOptions

- ParallelOptions
  - `MaxDegreeOfParallelism`
    Limits how many iterations run at the same time.
  - `CancellationToken`
    Cancel the loop before completion.
  - `TaskScheduler`
    Manage where the iterations run.

Producer / Consumer



What are Channels?

- Similar to a concurrent queue.
  - Write items to the channel.
  - Read items from the channel in the same order they were added.
  - Items are removed as they are read.
- Concurrent means that you can safely write and read from multiple threads without worry of missed writes or duplicate reads.

Where to Get Channels

- Built in to .NET 8 / 9 / 10
- .NET Framework NuGet Package
  - System.Threading.Channels

Parallel with Channels

- Overall Steps
  - Create a channel
  - Write to a channel
  - Read from a channel
  - Mark the channel “complete”

Creating a Channel

- CreateBounded<T>
  - Creates a channel of a specific size
  - If the channel is full, writers are blocked until space is available

```
var channel = Channel.CreateBounded<Person>(10);
```

Writing to a Channel

- writer.WriteAsync()
  - Writes an item to the channel

```
await writer.WriteAsync(person);
```

Reading from a Channel (.NET 8 / 9 / 10)

- reader.ReadAllAsync()
  - Returns an IAsyncEnumerable<T>

```
await foreach(var person in reader.ReadAllAsync())
{
   // use item here
}
```

  - If the channel is empty, the loop will pause until an item is available.
  - If the channel is "complete", the loop will exit.

Reading from a Channel (.NET Framework)

- WaitToReadAsync and TryRead can be used to retrieve items.

```
while (await reader.WaitToReadAsync())
{
    while (reader.TryRead(out Person person))
    {
        // use item here
    }
}
```

Marking a Channel “Complete”

- writer.Complete()
  - Indicates that no further items will be written
  - Writing to a “complete” channel throws an exception
  - Reading from a “complete” channel will continue normally until the channel is empty

Comparing Parallel Approaches

	Await	Task	ForEachAsync	Channel
Runs in Parallel	No	**Yes**	**Yes**	**Yes**
Continuation on Main Thread	**Yes**	**Yes** (optional)	No	**Yes**
Continuation in Parallel	No	**Yes**	**Yes**	No (optional)
Set Degrees of Parallelism	No	No	**Yes**	No

Parallel Considerations

- Thread-safe collections
- Thread-safe operations
- Continuing on the main thread
- Limit degrees of parallelism

Thread-safe Collections

- `List<T>` is not thread-safe. If you call `Add` from concurrent processes, some items may not be added.
- When writing to a collection from concurrent processes (such as parallel code), use a concurrent collection.

- System.Collections.Concurrent
- `BlockingCollection<T>` is a good place to start.

Thread-safe Increment

- Increment (++) and decrement (--) operators are not thread-safe.
- The `Interlocked` class has a useful set of thread-safe methods.
  - `Increment`
  - `Decrement`
  - `Add`
  - Several others

Interlocked.Increment Sample

```
await Parallel.ForEachAsync(items, async (i, _) => {
    try {
        // Process Item
        Interlocked.Increment(ref TotalProcessed);
    }
    catch (Exception ex) {
        Interlocked.Increment(ref TotalExceptions);
        ExceptionReporter.ShowException(ex);
    }
});
```

Continuations on Main Thread

- For Windows Desktop and MAUI applications, the Dispatcher class can be used to run a delegate on the UI thread.

Dispatcher Samples

- Windows Desktop - Dispatcher.Invoke(Action)

```
Dispatcher.Invoke(() => CreateUIElements(prediction));
```

- MAUI - Dispatcher.Dispatch(Action)

```
Dispatcher.Dispatch(() => CreateUIElements(prediction));
```

Max Degrees of Parallelism

- `SemaphoreSlim` can be used to limit the number of concurrent processes.

- Steps
  - Create `SemaphoreSlim` with initial count.
  - `await WaitAsync` to wait for entry.
  - Call `Release` when done.

Semaphore Slim Sample

```
using SemaphoreSlim semaphore = new(10);
foreach (var imageData in validation) {
    await semaphore.WaitAsync();
    var preditionTask = classifier.Predict(imageData);
    var continuation = predictionTask.ContinueWith(t => {
        CreateUIElements(t.Result);
        semaphore.Release();
    });
}
```

Comparing Parallel Approaches

	Await	Task	ForEachAsync	Channel
Runs in Parallel	No	**Yes**	**Yes**	**Yes**
Continuation on Main Thread	**Yes**	**Yes** (optional)	No	**Yes**
Continuation in Parallel	No	**Yes**	**Yes**	No (optional)
Set Degrees of Parallelism	No	No	**Yes**	No

Other Tools

- **Parallel.ForEach**
  Does not work with async in the iteration
- **PLINQ / .AsParallel()**
  Does not work with async in the iteration
- **DataFlow**
  https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/dataflow-task-parallel-library

Session Survey

- Your feedback is very important to us
- Please take a moment to complete the session survey found in the mobile app
- Use the QR code or search for “Converge360 Events” in your app store
- Find this session on the Agenda tab
- Click “Session Evaluation”
- Thank you!

LIVE! 360
TECH EVENTS WITH PERSPECTIVE

Thank You!

Jeremy Clark

- jeremybytes.com
- youtube.com/jeremybytes
- github.com/jeremybytes

https://github.com/jeremybytes/vslive2025-orlando